SECTION 1 — INTRODUCTION

OVERVIEW

This document is the reference manual for the POSTGRES95 database management system under development at the University of California at Berkeley. The POSTGRES95 project, led by Professor Michael Stonebraker, has been sponsored by the Defense Advanced Research Projects Agency (DARPA), the Army Research Office (ARO), the National Science Foundation (NSF), and Digital Equipment Corporation

This reference manual describes Version 1.01 of POSTGRES95. It assumes proficiency in C, C++, and SQL programming. POSTGRES95 is copyrighted by the Regents of the University of California. However, the source code is freely modifiable and distributable. Information on obtaining POSTGRES95 source code is available from the POSTGRES95 home page at the following URL:

http://s2k-ftp.cs.berkeley.edu:8000/postgres95

POSTGRES95 is based on the last official release of POSTGRES, Version 4.2 and provides enhanced performance and managibility. The code has been trimmed by 25% and runs 30-50% faster than POSTGRES v.4.2 on the Wisconsin Benchmark. More significantly POSTGRES95 no longer uses POSTQUEL query language but rather employs standard query language, SQL. For a more detailed description of specific changes to the code please consult the POSTGRES95 User's Manual.

POSTGRES95 has been complied and tested on

Architecture	Processor	Operating System
Intel x86	Pentium	Linux 1.3.42
DECstation 3000	Alpha AXP	OSF/2.0, 3.0
DECstation 5000	MIPS	ULTRIX 4.4
Sun4	SPARC	SunOS 4.1.3, 4.1.3_UI, Solaris 2.4
HP 9000/700 & 800	PA-RISC	HP-UX 9.00, 9.01, 9.03

Application developers should note that this manual contains only the specification for the low-level call-oriented application program interface, LIBPQ. A companion volume, the POSTGRES95 User's Manual, contains tutorial examples of the ways in which the system can be extended.

The remainder of this reference manual is structured as follows. Section 2 (UNIX), discusses the POSTGRES95 capabilities that are available directly from the operating system. Section 3 (BUILT-INS) describes POSTGRES95 internal data types, functions, and operators. Section 4 (COMMANDS) describes SQL, the language by which a user interacts with a POSTGRES95 database. Then, Section 5 (LIBPQ) describes a library of low level routines through which a user can formulate SQL queries from a C program and get appropriate return information back to his program. Section 6 (LARGE OBJECTS) describes the internal POSTGRES95 interface for accessing large objects. Section 7 (SYSTEM CATALOGS) gives a brief explanation of the tables used internally by POSTGRES95. The reference concludes with Section 8 (FILES), a collection of file format descriptions for files used by POSTGRES95.

ACKNOWLEDGEMENTS

The primary authors of POSTGRES95 are Jolly Chen and Andrew Yu. Previous versions of POSTGRES was constructed by a team of undergraduate, graduate, and staff programmers: Jeff Anton, Paul Aoki, James Bell, Jennifer Caetta, Philip Chang, Jolly Chen, Ron Choi, Matt Dillon, Zelaine Fong, Adam Glass, Jeffrey Goh, Steven Grady, Serge Granik, Marti Hearst, Joey Hellerstein, Michael Hirohama, Chin-heng Hong, Wei Hong, Anant Jhingran, Greg Kemnitz, Marcel Kornacker, Case Larsen, Boris Livshitz, Jeff Meredith, Ginger Ogle, Michael Olson, Nels Olson, Lay-Peng Ong, Carol Paxson, Avi Pfeffer, Spyros

Potamianos, Sunita Sarawagi, David Muir Sharnoff, Mark Sullivan, Cimarron Taylor, Marc Teitelbaum, Yongdong Wang, Kristin Wright and Andrew Yu.

This reference was collectively written by the above implementation team, assisted by Claire Mosher and Dorothy Moore.

LEGAL NOTICES

SECTION 2 — UNIX COMMANDS (UNIX)

OVERVIEW

This section outlines the interaction between POSTGRES95 and the operating system. In particular, this section describes the POSTGRES95 support programs that are executable as UNIX commands.

TERMINOLOGY

In the following documentation, the term *site* may be interpreted as the host machine on which POSTGRES95 is installed. Since it is possible to install more than one set of POSTGRES95 databases on a single host, this term more precisely denotes any particular set of installed POSTGRES95 binaries and databases.

The *POSTGRES95 super-user* is the user named “postgres” who owns the POSTGRES95 binaries and database files. As the database super-user, all protection mechanisms may be bypassed and any data accessed arbitrarily. In addition, the POSTGRES95 super-user is allowed to execute some support programs which are generally not available to all users. Note that the POSTGRES95 super-user is *not* the same as the UNIX super-user, *root*, and should have a non-zero userid for security reasons.

The *database base administrator* or DBA, is the person who is responsible for installing POSTGRES95 to enforce a security policy for a site. The DBA can add new users by the method described below and maintain a set of template databases for use by *createdb*(unix).

The *postmaster* is the process that acts as a clearing-house for requests to the POSTGRES95 system. Frontend applications connect to the *postmaster*, which keeps tracks of any system errors and communication between the backend processes. The *postmaster* can take several command-line arguments to tune its behavior. However, supplying arguments is necessary only if you intend to run multiple sites or a non-default site. See *postmaster*(unix) for details.

The *POSTGRES95 backend* (the actual executable program called "postgres") may be executed directly from the user shell by the POSTGRES95 super-user (with the database name as an argument). However, doing this bypasses the shared buffer pool and lock table associated with a postmaster/site, therefore this is not recommended in a multiuser site.

NOTATION

“.../” at the front of a file name is used to represent the path to the POSTGRES95 super-user’s home directory. Anything in brackets (“[” and “]”) is optional. Anything in braces (“{” and “}”) can be repeated 0 or more times. Parentheses (“(” and “)”) are used to group boolean expressions. “|” is the boolean operator OR .

USING POSTGRES95 FROM UNIX

All POSTGRES95 commands that are executed directly from a UNIX shell are found in the directory “.../bin”. Including this directory in your search path will make executing the commands easier.

A collection of system catalogs exist at each site. These include a class (“pg_user”) that contains an instance for each valid POSTGRES95 user. The instance specifies a set of POSTGRES95 privileges, such as the ability to act as POSTGRES95 super-user, the ability to create/destroy databases, and the ability to update the system catalogs. A UNIX user cannot do anything with POSTGRES95 until an appropriate instance is installed in this class. Further information on the system catalogs is available by running queries on the appropriate classes.

USER AUTHENTICATION

Authentication is the process by which the backend server and *postmaster* ensure that the user requesting access to data is in fact who he/she claims to be. All users who invoke POSTGRES95 are checked against the contents of the “pg_user” class to ensure that they are authorized to do so. However, verification of the user's actual identity is performed in a variety of ways.

From the user shell

A backend server started from a user shell notes the user's (effective) user-id before performing a *setuid*(3) to the user-id of user "postgres". The effective user-id is used as the basis for access control checks. No other authentication is conducted.

From the network

If the POSTGRES95 system is built as distributed, access to the Internet TCP port of the *postmaster* process is available to anyone. However, POSTGRES95 offers optional host-based authentication where only access from certain hosts are allowed. Of course, host-based authentication is not fool-proof in Unix, either. It is possible for determined intruders to also masquerade the origination host. Those security issues are beyond the scope of POSTGRES95.

If greater security is desired, POSTGRES95 and its clients may be modified to use a network authentication system. For example, the *postmaster*, *monitor, psql* and the *libpq* library have already been configured to use either Version 4 or Version 5 of the *Kerberos* authentication system from the Massachusetts Institute of Technology. For more information on using *Kerberos* with POSTGRES95, see the appendix below.

ACCESS CONTROL

POSTGRES95 provides mechanisms to allow users to limit the access to their data that is provided to other users.

Database superusers

Database super-users (i.e., users who have "pg_user.usesuper" set) silently bypass all of the access controls described below with two exceptions: manual system catalog updates are not permitted if the user does not have "pg_user.usecatupd" set, and destruction of system catalogs (or modification of their schemas) is never allowed.

Access Privilege

The use of access privilege to limit reading, writing and setting of rules on classes is covered in *grant/revoke*(commands).

Class removal and schema modification

Commands that destroy or modify the structure of an existing class, such as *alter*, *drop table*, and *drop index*, only operate for the owner of the class. As mentioned above, these operations are **never** permitted on system catalogs.

FUNCTIONS AND RULES

Functions and rules allow users to insert code into the backend server that other users may execute without knowing it. Hence, both mechanisms permit users to **trojan horse** others with relative impunity. The only real protection is tight control over who can define functions (e.g., write to relations with SQL fields) and rules. Audit trails and alerters on "pg_class", "pg_user" and "pg_group" are also recommended.

Functions

Functions written in any language except SQL run inside the backend server process with the permissions of the user "postgres" (the backend server runs with its real and effective user-id set to "postgres"). It is possible for users to change the server's internal data structures from inside of trusted functions. Hence, among many other things, such functions can circumvent any system access controls. This is an inherent problem with user-defined C functions.

Rules

Like SQL functions, rules always run with the identity and permissions of the user who invoked the backend server.

SEE ALSO

postmaster(unix), alter(commands), insert(commands), grant/revoke(commands), copy(commands), create(commands), delete(commands), drop table(commands), drop index(commands), drop rule(commands), update(commands), select(commands), kerberos(1), kinit(1), kerberos(3)

CAVEATS

There are no plans to explicitly support encrypted data inside of POSTGRES95 (though there is nothing to prevent users from encrypting data within user-defined functions). There are no plans to explicitly support encrypted network connections, either, pending a total rewrite of the frontend/backend protocol.

User names, group names and associated system identifiers (e.g., the contents of “pg_user.usesysid”) are assumed to be unique throughout a database. Unpredictable results may occur if they are not.

APPENDIX: USING KERBEROS

Availability

The *Kerberos* authentication system is not distributed with POSTGRES95, nor is it available from the University of California at Berkeley. Versions of *Kerberos* are typically available as optional software from operating system vendors. In addition, a source code distribution may be obtained through MIT Project Athena by anonymous FTP from ATHENA-DIST.MIT.EDU (18.71.0.38). (You may wish to obtain the MIT version even if your vendor provides a version, since some vendor ports have been deliberately crippled or rendered non-interoperable with the MIT version.) Users located outside the United States of America and Canada are warned that distribution of the actual encryption code in *Kerberos* is restricted by U. S. government export regulations.

Any additional inquiries should be directed to your vendor or MIT Project Athena (“info-kerberos@ATHENA.MIT.EDU”). Note that FAQLs (Frequently-Asked Questions Lists) are periodically posted to the *Kerberos* mailing list, “kerberos@ATHENA.MIT.EDU” (send mail to “kerberos-request@ATHENA.MIT.EDU” to subscribe), and USENET news group, “comp.protocols.kerberos”.

Installation

Installation of *Kerberos* itself is covered in detail in the *Kerberos Installation Notes*. Make sure that the server key file (the *srvtab* or *keytab*) is somehow readable by user “postgres”.

POSTGRES95 and its clients can be compiled to use either Version 4 or Version 5 of the MIT *Kerberos* protocols by setting the KRBVERS variable in the file “.../src/Makefile.global” to the appropriate value. You can also change the location where POSTGRES95 expects to find the associated libraries, header files and its own server key file.

After compilation is complete, POSTGRES95 must be registered as a *Kerberos* service. See the *Kerberos Operations Notes* and related manual pages for more details on registering services.

Operation

After initial installation, POSTGRES95 should operate in all ways as a normal *Kerberos* service. For details on the use of authentication, see the manual pages for *postmaster*(unix) and *monitor*(unix).

In the *Kerberos* Version 5 hooks, the following assumptions are made about user and service naming: (1) user principal names (anames) are assumed to contain the actual UNIX/POSTGRES95 user name in the first component; (2) the POSTGRES95 service is assumed to be have two components, the service name and a hostname, canonicalized as in Version 4 (i.e., all domain suffixes removed).

```
user example: frew@S2K.ORG
user example: aoki/HOST=miyu.S2K.Berkeley.EDU@S2K.ORG
host example: postgres_dbms/ucbvax@S2K.ORG
```

Support for Version 4 will disappear sometime after the production release of Version 5 by MIT.

NAME

cleardbdir — completely destroys all database files

SYNOPSIS

cleardbdir

DESCRIPTION

cleardbdir destroys all the database files. It is used only by the POSTGRES95 super-user before re-initializing the entire installation for a particular site. Normal database users should never use this command.

The POSTGRES95 super-user should ensure the *postmaster* process is not running before running cleardbdir.

SEE ALSO

initdb(unix)

NAME

createdb — create a database

SYNOPSIS

createdb [**-a** system] [**-h** host] [**-p** port] [dbname]

DESCRIPTION

Createdb creates a new database. The person who executes this command becomes the database administrator, or DBA, for this database and is the only person, other than the POSTGRES95 super-user, who can destroy it.

Createdb is a shell script that invokes the POSTGRES95 *monitor*. Hence, a *postmaster* process must be running on the database server host before *createdb* is executed. In addition, the PGOPTION and PGREALM environment variables will be passed on to *monitor* and processed as described in *monitor*(unix).

The optional argument *dbname* specifies the name of the database to be created. The name must be unique among all POSTGRES95 databases. *Dbname* defaults to the value of the USER environment variable.

Createdb understands the following command-line options:

-a system

Specifies an authentication system *system* (see *introduction*(unix)) to use in connecting to the *postmaster* process. The default is site-specific.

-h host

Specifies the hostname of the machine on which the *postmaster* is running. Defaults to the name of the local host, or the value of the PGHOST environment variable (if set).

-p port

Specifies the Internet TCP port on which the *postmaster* is listening for connections. Defaults to 5432, or the value of the PGPORT environment variable (if set).

EXAMPLES

```
# create 5432 demo database
createdb demo


# create the demo database using the postmaster on host eden,
# port using the Kerberos authentication system.
createdb -a kerberos -p 5432 -h eden demo
```

FILES

$PGDATA/base/*dbname*

The location of the files corresponding to the database *dbname*.

SEE ALSO

createdb(commands), destroydb(unix), initdb(unix), monitor(unix), postmaster(unix).

DIAGNOSTICS

Error: Failed to connect to backend (host=*xxx*, port=*xxx*)

Createdb could not attach to the *postmaster* process on the specified host and port. If you see this message, ensure that the *postmaster* is running on the proper host and that you have specified the proper port. If your site uses an authentication system, ensure that you have obtained the required authentication credentials.

user "*username*" is not in "pg_user"
You do not have a valid entry in the relation "pg_user" and cannot do anything with POSTGRES95 at all; contact your POSTGRES95 site administrator.

user "*username*" is not allowed to create/destroy databases
You do not have permission to create new databases; contact your POSTGRES95 site administrator.

dbname **already exists**
The database already exists.

database creation failed on *dbname*
An internal error occurred in *monitor* or the backend server. Ensure that your POSTGRES95 site administrator has properly installed POSTGRES95 and initialized the site with *initdb*.

NAME

createuser — create a POSTGRES95 user

SYNOPSIS

createuser [**-a** system] [**-h** host] [**-p** port] [username]

DESCRIPTION

Createuser creates a new POSTGRES95 user. Only users with “usesuper” set in the “pg_user” class can create new POSTGRES95 users. As shipped, the user “postgres” can create users.

Createuser is a shell script that invokes *monitor*. Hence, a *postmaster* process must be running on the database server host before *createuser* is executed. In addition, the PGOPTION and PGREALM environment variables will be passed on to *monitor* and processed as described in *monitor*(unix).

The optional argument *username* specifies the name of the POSTGRES95 user to be created. (The invoker will be prompted for a name if none is specified on the command line.) This name must be unique among all POSTGRES95 users.

Createuser understands the following command-line options:

-a system

Specifies an authentication system *system* (see *introduction*(unix)) to use in connecting to the *postmaster* process. The default is site-specific.

-h host

Specifies the hostname of the machine on which the *postmaster* is running. Defaults to the name of the local host, or the value of the PGHOST environment variable (if set).

-p port

Specifies the Internet TCP port on which the *postmaster* is listening for connections. Defaults to 5432, or the value of the PGPORT environment variable (if set).

INTERACTIVE QUESTIONS

Once invoked with the above options, *createuser* will ask a series of questions. The new users’s login name (if not given on the command line) and user-id must be specified. (Note that the POSTGRES95 user-id must be the same as the user’s UNIX user-id.) In addition, you must describe the security capabilities of the new user. Specifically, you will be asked whether the new user should be able to act as POSTGRES95 super-user, create new databases and update the system catalogs manually.

SEE ALSO

destroyuser(unix), monitor(unix), postmaster(unix).

DIAGNOSTICS

Error: Failed to connect to backend (host=*xxx*, port=*xxx*)

Createuser could not attach to the *postmaster* process on the specified host and port. If you see this message, ensure that the *postmaster* is running on the proper host and that you have specified the proper port. If your site uses an authentication system, ensure that you have obtained the required authentication credentials.

user “*username*” is not in “pg_user”

You do not have a valid entry in the relation “pg_user” and cannot do anything with POSTGRES95 at all; contact your POSTGRES95 site administrator.

username **cannot create users.**

You do not have permission to create new users; contact your POSTGRES95 site administrator.

user “*username*” already exists

The user to be added already has an entry in the “pg_user” class.

database access failed

An internal error occurred in *monitor* or the backend server. Ensure that your POSTGRES95 site administrator has properly installed POSTGRES95 and initialized the site with *initdb*.

BUGS

POSTGRES95 user-ids and user names should not have anything to do with the constraints of UNIX.

NAME

destroydb — destroy an existing database

SYNOPSIS

destroydb [**-a** system] [**-h** host] [**-p** port] [dbname]

DESCRIPTION

Destroydb destroys an existing database. To execute this command, the user must be the database administrator, or DBA, for this database. The program runs silently; no confirmation message will be displayed. After the database is destroyed, a UNIX shell prompt will reappear.

Destroydb is a shell script that invokes *monitor*. Hence, a *postmaster* process must be running on the database server host before *destroydb* is executed. In addition, the PGOPTION and PGREALM environment variables will be passed on to *monitor* and processed as described in *monitor*(unix).

The optional argument *dbname* specifies the name of the database to be destroyed. All references to the database are removed, including the directory containing this database and its associated files. *Dbname* defaults to the value of the USER environment variable.

Destroydb understands the following command-line options:

-a system
: Specifies an authentication system *system* (see *introduction*(unix)) to use in connecting to the *postmaster* process. The default is site-specific.

-h host
: Specifies the hostname of the machine on which the *postmaster* is running. Defaults to the name of the local host, or the value of the PGHOST environment variable (if set).

-p port
: Specifies the Internet TCP port on which the *postmaster* is listening for connections. Defaults to 5432, or the value of the PGPORT environment variable (if set).

EXAMPLES

```
# destroy the demo database
destroydb demo


# destroy 5432 demo database using the postmaster on host eden,
# port using the Kerberos authentication system.
destroydb -a kerberos -p 5432 -h eden demo
```

FILES

$PGDATA/base/*dbname*
: The location of the files corresponding to the database *dbname*.

SEE ALSO

destroydb(commands), createdb(unix), initdb(unix), monitor(unix). postmaster(unix).

DIAGNOSTICS

Error: Failed to connect to backend (host=*xxx*, port=*xxx*)
: *Destroydb* could not attach to the *postmaster* process on the specified host and port. If you see this message, ensure that the *postmaster* is running on the proper host and that you have specified the proper port. If your site uses an authentication system, ensure that you have obtained the required authentication credentials.

user "*username*" is not in "pg_user"
You do not have a valid entry in the relation "pg_user" and cannot do anything with POSTGRES95 at all; contact your POSTGRES95 site administrator.

user "*username*" is not allowed to create/destroy databases
You do not have permission to destroy databases; contact your POSTGRES95 site administrator.

database "dbname" does not exist
The database to be removed does not have an entry in the "pg_database" class.

database "*dbname*" is not owned by you
You are not DBA for the specified database.

database destroy failed on *dbname*
An internal error occurred in *monitor* or the backend server. Contact your POSTGRES95 site administrator to ensure that ensure that the files and database entries associated with the database are completely removed.

NAME

destroyuser — destroy a POSTGRES95 user and associated databases

SYNOPSIS

destroyuser [**-a** system] [**-h** host] [**-p** port] [username]

DESCRIPTION

Destroyuser destroys an existing POSTGRES95 user and the databases for which that user is database administrator. Only users with “usesuper” set in the “pg_user” class can destroy new POSTGRES95 users. As shipped, the user “postgres” can destroy users.

Destroyuser is a shell script that invokes *monitor*. Hence, a *postmaster* process must be running on the database server host before *destroyuser* is executed. In addition, the PGOPTION and PGREALM environment variables will be passed on to *monitor* and processed as described in *monitor*(unix).

The optional argument *username* specifies the name of the POSTGRES95 user to be destroyed. (The invoker will be prompted for a name if none is specified on the command line.)

Destroyuser understands the following command-line options:

-a system
: Specifies an authentication system *system* (see *introduction*(unix)) to use in connecting to the *postmaster* process. The default is site-specific.

-h host
: Specifies the hostname of the machine on which the *postmaster* is running. Defaults to the name of the local host, or the value of the PGHOST environment variable (if set).

-p port
: Specifies the Internet TCP port on which the *postmaster* is listening for connections. Defaults to 5432, or the value of the PGPORT environment variable (if set).

INTERACTIVE QUESTIONS

Once invoked with the above options, *destroyuser* will warn you about the databases that will be destroyed in the process and permit you to abort the removal of the user if desired.

SEE ALSO

createuser(unix), monitor(unix), postmaster(unix).

DIAGNOSTICS

Error: Failed to connect to backend (host=*xxx*, port=*xxx*)
: *Destroyuser* could not attach to the *postmaster* process on the specified host and port. If you see this message, ensure that the *postmaster* is running on the proper host and that you have specified the proper port. If your site uses an authentication system, ensure that you have obtained the required authentication credentials.

user “*username*” is not in “pg_user”
: You do not have a valid entry in the relation “pg_user” and cannot do anything with POSTGRES95 at all; contact your POSTGRES95 site administrator.

username **cannot delete users.**
: You do not have permission to delete users; contact your POSTGRES95 site administrator.

user “*username*” does not exist
: The user to be removed does not have an entry in the “pg_user” class.

database access failed

destroydb on *dbname* **failed - exiting**

delete of user *username* **was UNSUCCESSFUL**

An internal error occurred in *monitor* or the backend server. Contact your POSTGRES95 site administrator to ensure that the files and database entries associated with the user and his/her associated databases are completely removed.

NAME

initdb — initalize the database templates and primary directories

SYNOPSIS

initdb [**-v**] [**-d**] [**-n**]

DESCRIPTION

Initdb sets up the initial template databases and is normally executed as part of the installation process. The template database is created under the directory specified by the the environment variable PGDATA, or to a default specified at compile-time. The template database is then **vacuum**ed.

Initdb is a shell script that invokes the backend server directly. Hence, it must be executed by the POSTGRES95 super-user.

Initdb understands the following command-line options:

-v Produce verbose output, printing messages stating where the directories are being created, etc.

-d Print debugging output from the backend server. This option generates a tremendous amount of information. This option also turns off the final vacuuming step.

-n Run in "noclean" mode. By default, *initdb* cleans up (recursively unlinks) the data directory if any error occurs, which also removes any core files left by the backend server. This option inhibits any tidying-up.

FILES

$PGDATA/base
: The location of global (shared) classes.

$PGDATA/base/template1
: The location of the template database.

$PGDATA/files/{global1,local1_template1}.bki
: Command files used to generate the global and template databases, generated and installed by the initial compilation process.

SEE ALSO

createdb(unix), vacuum(commands), bki(files), template(files).

NAME

ipcclean — clean up shared memory and semaphores from aborted backends

SYNOPSIS

ipcclean

DESCRIPTION

Ipcclean cleans up shared memory and semaphore space from aborted backends by deleting all instances owned by user "postgres". Only the DBA should execute this program as it can cause bizarre behavior (i.e., crashes) if run during multi-user execution. This program should be executed if messages such as **semget: No space left on device** are encountered when starting up the *postmaster* or the backend server.

BUGS

If this command is executed while a *postmaster* is running, the shared memory and semaphores allocated by the *postmaster* will be deleted. This will result in a general failure of the backends servers started by that *postmaster*.

This script is a hack, but in the many years since it was written, no one has come up with an equally effective and portable solution. Suggestions are welcome.

The script makes assumption about the format of output of the **ipcs** utility which may not be true across different operating systems. Therefore, it may not work on your particular OS.

NAME

monitor — run the interactive terminal monitor

SYNOPSIS

monitor [**-N**] [**-Q**] [**-T**] [**-a** system] [**-c** query] [**-d** path]
[**-h** hostname] [**-p** port] [**-q**] [**-t** tty_device] [dbname]

DESCRIPTION

The interactive terminal monitor is a simple frontend to POSTGRES95 retained for backwards compatiblity. Users are encouraged to the use the *psql* interface instead.

monitor enables you to formulate, edit and review queries before issuing them to POSTGRES95. If changes must be made, a UNIX editor may be called to edit the **query buffer** managed by the terminal monitor. The editor used is determined by the value of the EDITOR environment variable. If EDITOR is not set, then **vi** is used by default.

Monitor is a frontend application, like any other. Hence, a *postmaster* process must be running on the database server host before *monitor* is executed. In addition, the correct *postmaster* port number must be specified as described below.

The optional argument *dbname* specifies the name of the database to be accessed. This database must already have been created using *createdb*. *Dbname* defaults to the value of the USER environment variable.

Monitor understands the following command-line options:

-N Specifies that query results will be dumped to the screen without any attempt at formatting. This is useful in with the **-c** option in shell scripts.

-Q Produces extremely unverbose output. This is useful with the **-c** option in shell scripts.

-T Specifies that attribute names will not be printed. This is useful with the **-c** option in shell scripts.

-a system
Specifies an authentication system *system* (see *introduction*(unix)) to use in connecting to the *postmaster* process. The default is site-specific.

-c query
Specifies that *monitor* is to execute one query string, *query*, and then exit. This is useful for shell scripts, typically in conjunction with the **-N** and **-T** options. Examples of shell scripts in the POSTGRES95 distribution using *monitor* **-c** include *createdb*, *destroydb*, *createuser*, and *destroyuser*,

-d path
path specifies the path name of the file or tty to which frontend (i.e., *monitor*) debugging messages are to be written; the default is not to generate any debugging messages.

-h hostname
Specifies the hostname of the machine on which the *postmaster* is running. Defaults to the name of the local host, or the value of the PGHOST environment variable (if set).

-p port
Specifies the Internet TCP port on which the *postmaster* is listening for connections. Defaults to 5432, or the value of the PGPORT environment variable (if set).

-q Specifies that the monitor should do its work quietly. By default, it prints welcome and exit messages and the queries it sends to the backend. If this option is used, none of this happens.

-t tty_device
tty_device specifies the path name to the file or tty to which backend (i.e., *postgres*) debugging messages are to be written; the default is */dev/null*.

-s parses after each query (enables "single step" mode).

-S Turns off sending query when ";" is encountered.

You may set environment variables to avoid typing some of the above options. See the ENVIRONMENT VARIABLES section below.

MESSAGES AND PROMPTS

The terminal monitor gives a variety of messages to keep the user informed of the status of the monitor and the query buffer.

The terminal monitor displays two kinds of messages:

go The query buffer is empty and the terminal monitor is ready for input. Anything typed will be added to the buffer.

* This prompt is typed at the beginning of each line when the terminal monitor is waiting for input.

TERMINAL MONITOR COMMANDS

\e Enter the editor to edit the query buffer.

\g Submit query buffer to POSTGRES95 for execution.

\h Get on-line help.

\i *filename*
Include the file *filename* into the query buffer.

\p Print the current contents of the query buffer.

\q Exit from the terminal monitor.

\r Reset (clear) the query buffer.

\s Escape to a UNIX subshell. To return to the terminal monitor, type “exit” at the shell prompt.

\t Print the current time.

\w *filename*
Store (write) the query buffer to an external file *filename*.

\\ Produce a single backslash at the current location in query buffer.

\; Produce a single semi-colon at the current location in query buffer.

ENVIRONMENT VARIABLES

You may set any of the following environment variables to avoid specifying command-line options:

hostname:	PGHOST
port:	PGPORT
tty:	PGTTY
options:	PGOPTION
realm:	PGREALM

If PGOPTION is specified, then the options it contains are parsed **before** any command-line options.

PGREALM only applies if *Kerberos* authentication is in use. If this environment variable is set, POSTGRES95 will attempt authentication with servers for this realm and use separate ticket files to avoid conflicts with local ticket files. See *introduction*(unix) for additional information on *Kerberos*.

See *introduction*(libpq) for additional details.

RETURN VALUE

When executed with the **-c** option, *monitor* returns 0 to the shell on successful query completion, 1 otherwise.

SEE ALSO

introduction(libpq), createdb(unix), createuser(unix), postgres(unix), postmaster(unix).

BUGS

Does not poll for asynchronous notification events generated by *listen*(commands) and *notify*(commands).

Escapes (backslash characters) cannot be commented out.

SEE ALSO

psql(unix)

NAME

pg_dump — dumps out a POSTGRES95 database into a script file

SYNOPSIS

pg_dump [**-f** filename] [**-H** hostname] [**-p** port] [**-v**] [**-S** help] dbname

DESCRIPTION

pg_dump is a utility for dumping out a POSTGRES95 database into a script file containing query commands. The script files are in a ASCII format and can be used to reconstruct the database, even on other machines and other architectures. *pg_dump* will produce the queries necessary to re-generate all user-defined types, functions, tables, indices, aggregates, and operators. In addition, all the data is copied out in ASCII format so that it can be readily copied in again, as well, as imported into tools for textual editing.

pg_dump is useful for dumping out the contents of a database to move from one postgres95 installation to another. After running *pg_dump* , one should examine the output script file for any warnings, especially in light of the limitations listed below.

CAVEATS AND LIMITATIONS

pg_dump has a few limitations. The limitations mostly stem from difficulty in extracting certain meta-information from the system catalogs.

rules and views

pg_dump does not understand user-defined rules and views and will fail to dump them properly. (This is due to the fact that rules are stored as plans in the catalogs and not textually)

partial indices

pg_dump does not understand partial indices. (The reason is the same as above. Partial index predicates are stored as plans)

large objects

pg_dump does not handle large objects. Large objects are ignored and must be dealt with manually.

oid preservation

pg_dump does not preserve oid's while dumping. If you have stored oid's explicitly in tables in user-defined attributes, and are using them as keys, then the output scripts will not regenerate your database correctly.

SEE ALSO

copy(commands)

NAME

postgres — the POSTGRES95 backend server

SYNOPSIS

postgres [**-B** n_buffers] [**-E**] [**-P** filedes] [**-Q**]
[**-d** debug_level] [**-o** output_file] [**-s**] [dbname]

DESCRIPTION

The POSTGRES95 backend server can be executed directly from the user shell. This should be done only while debugging by the DBA, and should not be done while other POSTGRES95 backends are being managed by a *postmaster* on this set of databases.

The optional argument *dbname* specifies the name of the database to be accessed. *Dbname* defaults to the value of the USER environment variable.

The *postgres* server understands the following command-line options:

-B n_buffers
If the backend is running under the *postmaster*, *n_buffers* is the number of shared-memory buffers that the *postmaster* has allocated for the backend server processes that it starts. If the backend is running standalone, this specifies the number of buffers to allocate. This value defaults to 64.

-E Echo all queries.

-P filedes
filedes specifies the file descriptor that corresponds to the socket (port) on which to communicate to the frontend process. This option is **not** useful for interactive use.

-Q Specifies “quiet” mode.

-d debug_level
Turns on debugging at the numeric level *debug_level*. Turning on debugging will cause query parse trees and query plans to be displayed.

-o output_file
Sends all debugging and error output to *output_file*. If the backend is running under the *postmaster*, error messages are still sent to the frontend process as well as to *output_file*, but debugging output is sent to the controlling tty of the *postmaster* (since only one file descriptor can be sent to an actual file).

-s Print time information and other statistics at the end of each query. This is useful for benchmarking or for use in tuning the number of buffers.

DEPRECATED COMMAND OPTIONS

There are several other options that may be specified, used mainly for debugging purposes. These are listed here only for the use by POSTGRES95 system developers. **Use of any of these options is highly discouraged**. Furthermore, any of these options may disappear or change at any time.

-An|r|b|Q*n*|X*n*

This option generates a tremendous amount of output.

-L Turns off the locking system.

-N Disables use of newline as a query delimiter.

-S Indicates that the transaction system can run with the assumption of stable main memory, thereby avoiding the necessary flushing of data and log pages to disk at the end of each transaction system. This is only used for performance comparisons for stable vs. non-stable storage. Do not use this in other cases, as recovery after a system crash may be impossible when this option is specified in the absence of stable main memory.

-b Enables generation of bushy query plan trees (as opposed to left-deep query plans trees). These query plans are not intended for actual execution; in addition, this flag often causes POSTGRES95 to run out of memory.

-f Forbids the use of particular scan and join methods: *s* and *i* disable sequential and index scans respectively, while *n*, *m* and *h* disable nested-loop, merge and hash joins respectively. This is another feature that may not necessarily produce executable plans.

-p Indicates to the backend server that it has been started by a *postmaster* and make different assumptions about buffer pool management, file descriptors, etc.

-tpa[rser]|pl[anner]|e[xecutor]
Print timing statistics for each query relating to each of the major system modules. This option cannot be used with **-s**.

SEE ALSO

ipcclean(unix), monitor(unix), postmaster(unix).

DIAGNOSTICS

Of the nigh-infinite number of error messages you may see when you execute the backend server directly, the most common will probably be:

semget: No space left on device
If you see this message, you should run the *ipcclean* command. After doing this, try starting *postgres* again. If this still doesn't work, you probably need to configure your kernel for shared memory and semaphores as described in the installation notes.

NAME

postmaster — run the POSTGRES95 postmaster

SYNOPSIS

postmaster [**-B** n_buffers] [**-D** data_dir] [**-S**] [**-a** system]
[**-b** backend_pathname] [**-d** [debug_level]] [**-n**]
[**-o** backend_options] [**-p** port] [**-s**]

DESCRIPTION

The *postmaster* manages the communication between frontend and backend processes, as well as allocating the shared buffer pool and semaphores (on machines without a test-and-set instruction). The *postmaster* does not itself interact with the user and should be started as a background process. **Only one postmaster should be run on a machine.**

The *postmaster* understands the following command-line options:

-B n_buffers
n_buffers is the number of shared-memory buffers for the *postmaster* to allocate and manage for the backend server processes that it starts. This value defaults to 64.

-D data_dir
Specifies the directory to use as the root of the tree of database directories. This directory uses the value of the environment variable PGDATA. If PGDATA is not set, then the directory used is $POSTGRESHOME/data. If neither environment variable is set and this command-line option is not specified, the default directory that was set at compile-time is used.

-S Specifies that the *postmaster* process should start up in silent mode. That is, it will disassociate from the user's (controlling) tty and start its own process group. This should not be used in combination with debugging options because any messages printed to standard output and standard error are discarded.

-a system
Specifies whether or not to use the authentication system *system* (see *introduction*(unix)) for frontend applications to use in connecting to the *postmaster* process. Specify *system* to enable a system, or **no***system* to disable a system. For example, to permit users to use *Kerberos* authentication, use **-a kerberos**; to deny any unauthenticated connections, use **-a nounauth .** The default is site-specific.

-b backend_pathname
backend_pathname is the full pathname of the POSTGRES95 backend server executable file that the *postmaster* will invoke when it receives a connection from a frontend application. If this option is not used, then the *postmaster* tries to find this executable file in the directory in which its own executable is located (this is done by looking at the pathname under which the *postmaster* was invoked. If no pathname was specified, then the PATH environment variable is searched for an executable named "postgres").

-d [debug_level]
The optional argument *debug_level* determines the amount of debugging output the backend servers will produce. If *debug_level* is one, the postmaster will trace all connection traffic, and nothing else. For levels two and higher, debugging is turned on in the backend process and the postmaster displays more information, including the backend environment and process traffic. Note that if no file is specified for backend servers to send their debugging output then this output will appear on the controlling tty of their parent *postmaster*.

-n, **-s**
The *-s* and *-n* options control the behavior of the *postmaster* when a backend dies abnormally. **Neither option is intended for use in ordinary operation**.

The ordinary strategy for this situation is to notify all other backends that they must terminate and then reinitialize the shared memory and semaphores. This is because an errant backend could have corrupted some shared state before terminating.

If the *-s* option is supplied, then the *postmaster* will stop all other backend processes by sending the signal SIGSTOP, but will not cause them to terminate. This permits system programmers to collect core dumps from all backend processes by hand.

If the *-n* option is supplied, then the *postmaster* does not reinitialize shared data structures. A knowledgable system programmer can then use the *shmemdoc* program to examine shared memory and semaphore state.

-o backend_options
The *postgres*(unix) options specified in *backend_options* are passed to all backend server processes started by this *postmaster*. If the option string contains any spaces, the entire string must be quoted.

-p port
Specifies the Internet TCP port on which the *postmaster* is to listen for connections from frontend applications. Defaults to 5432, or the value of the PGPORT environment variable (if set). If you specify a port other than the default port then all frontend application users must specify the same port (using command-line options or PGPORT) when starting any libpq application, including psql.

WARNINGS

If at all possible, **do not** use SIGKILL when killing the *postmaster*. SIGHUP, SIGINT, or SIGTERM (the default signal for *kill*(1)) should be used instead. Hence, avoid

```
kill -KILL
```

or its alternative form

```
kill -9
```

as this will prevent the *postmaster* from freeing the system resources (e.g., shared memory and semaphores) that it holds before dying. This prevents you from having to deal with the problem with *shmat*(2) described below.

EXAMPLES

```
# start postmaster using default values
postmaster -S
```

This command will start up *postmaster* on the default port (5432) and will search $PATH to find an executable file called “postgres”. This is the simplest and most common way to start the *postmaster*.

```
# start with specific port and executable name
postmaster -p -b /usr/postgres/bin/postgres &
```

This command will start up a *postmaster* communicating through the port 1234, and will attempt to use the backend located at “/usr/postgres/bin/postgres”. In order to connect to this *postmaster* using psql, you would need to either specify **-p 1234** on the *psql* command-line or set the environment variable PGPORT to 1234.

SEE ALSO

ipcs(1), ipcrm(1), ipcclean(unix), psql(unix), postgres(unix),

DIAGNOSTICS

FindBackend: could not find a backend to execute...

If you see this message, you do not have the *postgres* executable in your path. Add the directoy in which postgres resides to your path.

semget: No space left on device

If you see this message, you should run the *ipcclean* command. After doing this, try starting the *postmaster* again. If this still doesn't work, you probably need to configure your kernel for shared memory and semaphores as described in the installation notes. If you run multiple *postmaster*s on a single host, or have reduced the shared memory and semaphore parameters from the defaults in the generic kernel, you may have to go back and increase the shared memory and semaphores configured into your kernel.

StreamServerPort: cannot bind to port

If you see this message, you should be certain that there is no other *postmaster* process already running. The easiest way to determine this is by using the command

```
ps −ax | grep postmaster
```

on BSD-based systems

```
ps −e | grep postmast
```

(the equivalent syntax is on System V-like or POSIX-compliant systems such as HP-UX). If you are sure that no other *postmaster* processes are running and you still get this error, try specifying a different port using the **-p** option. You may also get this error if you terminate the *postmaster* and immediately restart it using the same port; in this case, you must simply wait a few seconds until the operating system closes the port before trying again. Finally, you may get this error if you specify a port number that your operating system considers to be reserved. For example, many versions of UNIX consider port numbers under 1024 to be "trusted" and only permit the UNIX superuser to access them.

IpcMemoryAttach: shmat() failed: Permission denied

A likely explanation is that another user attempted to start a *postmaster* process on the same port which acquired shared resources and then died. Since POSTGRES95 shared memory keys are based on the port number assigned to the *postmaster*, such conflicts are likely if there is more than one installation on a single host. If there are no other *postmaster* processes currently running (see above), run *ipcclean* and try again. If other *postmaster*s are running, you will have to find the owners of those processes to coordinate the assignment of port numbers and/or removal of unused shared memory segments.

NAME

psql — run the interactive query front-end

SYNOPSIS

psql [**-a** authsvc] [**-A**] [**-c** query] [**-d** dbName] [**-e**] [**-f** filename] [**-h** help]
[**-H** hostname] [**-l** port] [**-n**] [**-o** filename] [**-p** port] [**-q**] [**-s**] [**-S**] [**-T**] [dbname]

DESCRIPTION

psql is a interactive query front-end to POSTGRES95. It enables you to type in queries interactively, issue them to POSTGRES95, and see the query results. It is designed to be an enhanced version of the older *monitor* program.

psql is a frontend application, like any other. Hence, a *postmaster* process must be running on the database server host before *psql* is executed. In addition, the correct *postmaster* port number must be specified as described below.

The optional argument *dbname* specifies the name of the database to be accessed. This database must already have been created. *dbname* defaults to the value of the USER environment variable or, if that's not set, to the Unix account name of the current user.

psql understands the following command-line options:

-a system
: Specifies an authentication system *system* (see *introduction*(unix)) to use in connecting to the *postmaster* process. The default is site-specific.

-A Turn off fill justification when printing out attributes.

-c query
: Specifies that *psql* is to execute one query string, *query*, and then exit. This is useful for shell scripts, typically in conjunction with the **-q** options. **-c** option in shell scripts.

-d dbName
: Specifies the name of the database to connect to.

-e Echo the query sent to the backend

-f filename
: Use the file *filename* as the source of queries instead of reading queries interactively.

-h Give help information about psql.

-H hostname
: Specifies the hostname of the machine on which the *postmaster* is running. Defaults to the name of the local host, or the value of the PGHOST environment variable (if set).

-l Lists all available databases

-n Do not use the readline library for input line editing and command history.

-p port Specifies the Internet TCP port on which the *postmaster* is listening for connections. Defaults to 5432, or the value of the PGPORT environment variable (if set).

-q Specifes that *psql* should do its work quietly. By default, it prints welcome and exit messages and prompts for each query, and prints out the number of rows returned from a query. If this option is used, none of this happens. This is useful with the **-c** option in shell scripts.

-s Run in single-step mode where the user at prompted for each query before it is sent to the backend.

-S Run ins single-line mode where each query is terminated by a newline, instead of a semicolon.

-T Turn off printing of attributes names. This is useful with the **-c** option in shell scripts.

You may set environment variables to avoid typing some of the above options. See the ENVIRONMENT VARIABLES section below.

CONNECTING TO A DATABASE

psql attempts to make a connection to the database at the hostname and port number specified on the command line. If the connection could not be made for any reason (e.g. insufficient privileges, postmaster is not running on the server, etc) *psql* will return an error that says

```
Connection to database failed.
```

The reason for the connection failure is not provided.

ENTERING QUERIES

In normal operation, psql provides a prompt with the name of the database that psql is current connected to followed by the string "=>". For example,

```
Welcome to the POSTGRES95 interactive sql monitor:
  Please read the file COPYRIGHT for copyright terms of POSTGRES95

   type \? for help on slash commands
   type \q to quit
   type \g or terminate with semicolon to execute query
 You are currently connected to the database: testdb

testdb=>
```

At the prompt, the user may type in SQL queries. Unless the -S option is set, input lines are sent to the backend when a query-terminating semicolon is reached.

Whenever a query is executed, psql also polls for asynchronous notification events generated by *listen*(commands) and *notify*(commands).

PSQL COMMANDS

\a Toggle fill justification when printing out attributes.

\c *dbname*
Establish a connection to a new database. The previous connection is closed.

\d [*table*]
List tables in the database, or if *table* is specified, list the columns in *table.* If table name is *, list all tables and column information for each tables.

\f *separator*
Set the field separator. Default is a single blank space.

\h *command*
Give syntax help on the specified SQL command. If the *command* is not specified, list all the commands for which syntax help is available. If the *command* is *, give syntax help on all SQL commands.

\g Send the current query input buffer to the backend.

\i *filename*
Read queries from *filename* into the query input buffer.

\l List all the databases in the server.

\o *filename*
Send query results to *filename.* If *filename* is not specified, send query results to *stdout.*

\p Print the current query buffer.

\q Quit the psql program.

\s *history*
Save the command line history to a file. (Only available if psql is configured to use readline)

\t Toggle display of output attribute name headers (defaults to on).

\! *command*
Escape to shell and execute *command.*

\? Get help information about the \ commands.

ENVIRONMENT VARIABLES

You may set any of the following environment variables to avoid specifying command-line options:

hostname:	PGHOST
port:	PGPORT
tty:	PGTTY
options:	PGOPTION
realm:	PGREALM

If PGOPTION is specified, then the options it contains are parsed **before** any command-line options.

PGREALM only applies if *Kerberos* authentication is in use. If this environment variable is set, POSTGRES95 will attempt authentication with servers for this realm and use separate ticket files to avoid conflicts with local ticket files. See *introduction*(unix) for additional information on *Kerberos*.

See *introduction*(libpq) for additional details.

RETURN VALUE

When executed with the **-c** option, *psql* returns 0 to the shell on successful query completion, 1 otherwise. *psql* will also return 1 if the connection to a database could not be made for any reason.

SEE ALSO

introduction(libpq), monitor(unix) postgres(unix), postmaster(unix).

BUGS

If multiple queries are sent to the backend at once without semicolon termination after each query, psql gets confused about the query results. The queries will still be processed correctly by the backend.

SECTION 3 — WHAT COMES WITH POSTGRES95 (BUILT-INS)

DESCRIPTION

This section describes the data types, functions and operators available to users in POSTGRES95 as it is distributed.

BUILT-IN TYPES

This section describes both **built-in** data types. These Built-in types are are installed in every database.

Users may add new types to POSTGRES95 using the *define type* command described in this manual. User-defined types are not described in this section.

POSTGRES Type	Meaning
abstime	absolute date and time
aclitem	access control list item
bool	boolean
box	2-dimensional rectangle
bpchar	blank-padded characters
bytea	variable length array of bytes
char	character
char2	array of 2 characters
char4	array of 4 characters
char8	array of 8 characters
char16	array of 16 characters
cid	command identifier type
date	ANSI SQL date type
filename	large object filename
int	alias for int4
integer	alias for int4
int2	two-byte signed integer
int28	array of 8 int2
int4	four-byte signed integer
float	alias for float4
float4	single-precision floating-point number
float8	double-precision floating-point number
lseg	2-dimensional line segment
name	a multi-character type for storing system identifiers
oid	object identifier type
oid8	array of 8 oid
oidchar16	oid and char16 composed
oidint2	oid and int2 composed
oidint4	oid and int4 composed
path	variable-length array of lseg
point	2-dimensional geometric point
polygon	2-dimensional polygon
real	alias for float4
regproc	registered procedure
reltime	relative date and time
smgr	storage manager
smallint	alias for int2

text	variable length array of characters
tid	tuple identifier type
time	ANSI SQL time type
tinterval	time interval
varchar	variable-length characters
xid	transaction identifier type

As a rule, the built-in types are all either (1) internal types, in which case the user should not worry about their external format, or (2) have obvious formats. The exceptions to this rule are the three time types.

ABSOLUTE TIME

Absolute time is specified using the following syntax:

```
Month  Day [ Hour : Minute : Second ]  Year [ Timezone ]
```

where Month is Jan, Feb, ..., Dec
Day is 1, 2, ..., 31
Hour is 01, 02, ..., 24
Minute is 00, 01, ..., 59
Second is 00, 01, ..., 59
Year is 1901, 1902, ..., 2038

Valid dates are from Dec 13 20:45:53 1901 GMT to Jan 19 03:14:04 2038 GMT. As of Version 3.0, times are no longer read and written using Greenwich Mean Time; the input and output routines default to the local time zone.

The special absolute time values “current”, “infinity” and “-infinity” are also provided. “infinity” specifies a time later than any valid time, and “-infinity” specifies a time earlier than any valid time. “current” indicates that the current time should be substituted whenever this value appears in a computation.

The strings “now” and “epoch” can be used to specify time values. “now” means the current time, and differs from “current” in that the current time is immediately substituted for it. “epoch” means Jan 1 00:00:00 1970 GMT.

RELATIVE TIME

Relative time is specified with the following syntax:

```
@ Quantity Unit [Direction]
```

where Quantity is ‘1’, ‘2’, ...
Unit is ‘‘second’’, ‘‘minute’’, ‘‘hour’’, ‘‘day’’, ‘‘week’’,
‘‘month’’ (30-days), or ‘‘year’’ (365-days),
or PLURAL of these units.
Direction is ‘‘ago’’

(**Note**: Valid relative times are less than or equal to 68 years.) In addition, the special relative time “Undefined RelTime” is provided.

TIME RANGES

Time ranges are specified as:

```
[ 'abstime' 'abstime']
```

where *abstime* is a time in the absolute time format. Special abstime values such as “current”, “infinity” and “-infinity” can be used.

OPERATORS

POSTGRES95 provides a large number of built-in operators on system types. These operators are declared in the system catalog "pg_operator". Every entry in "pg_operator" includes the object ID of the procedure that implements the operator.

Users may invoke operators using the operator name, as in

```
select * from emp where salary < 40000;
```

Alternatively, users may call the functions that implement the operators directly. In this case, the query above would be expressed as

```
select * from emp where int4lt(salary, 40000);
```

The rest of this section provides a list of the built-in operators and the functions that implement them. Binary operators are listed first, followed by unary operators.

BINARY OPERATORS

This list was generated from the POSTGRES95 system catalogs with the query

```
select t1.typname as argtype,
      o.oprname, t0.typname,
      p.proname,
      t1.typname as ltype,
      t2.typname as rtype
    from pg_proc p, pg_type t0,
      pg_type t1, pg_type t2,
      pg_operator o
    where p.prorettype = t0.oid
      and RegprocToOid(o.oprcode) = p.oid
      and p.pronargs = 2
      and o.oprleft = t1.oid
      and o.oprright = t2.oid;
```

The list is sorted by the built-in type name of the first operand. The *function prototype* column gives the return type, function name, and argument types for the procedure that implements the operator. Note that these function prototypes are cast in terms of SQL types and so are **not** directly usable as C function prototypes.

Type	Operator	POSTGRES Function Prototype	Operation
abstime	<>	bool abstimene(abstime, abstime)	inequality
	+	abstime timepl(abstime, reltime)	addition
	–	abstime timemi(abstime, reltime)	subtraction
	<=	bool abstimele(abstime, abstime)	less or equal
	<?>	bool ininterval(abstime, tinterval)	abstime in tinterval?
	<	bool abstimelt(abstime, abstime)	less than
	=	bool abstimeeq(abstime, abstime)	equality
	>=	bool abstimege(abstime, abstime)	greater or equal
	>	bool abstimegt(abstime, abstime)	greater than
bool	=	bool booleq(bool, bool)	equality

	<>	bool boolne(bool, bool)	inequality
box	&&	bool box_overlap(box, box)	boxes overlap
	&<	bool box_overleft(box, box)	box A overlaps box B, but does not extend to right of box B
	&>	bool box_overright(box, box)	box A overlaps box B, but does not extend to left of box B
	<<	bool box_left(box, box)	A is left of B
	<=	bool box_le(box, box)	area less or equal
	<	bool box_lt(box, box)	area less than
	=	bool box_eq(box, box)	area equal
	>=	bool box_ge(box, box)	area greater or equal
	>>	bool box_right(box, box)	A is right of B
	>	bool box_gt(box, box)	area greater than
	@	bool box_contained(box, box)	A is contained in B
	~=	bool box_same(box, box)	box equality
	~	bool box_contain(box, box)	A contains B
char	<>	bool charne(char, char)	inequality
	*	bool charmul(char, char)	multiplication
	+	bool charpl(char, char)	addition
	−	bool charmi(char, char)	subtraction
	/	bool chardiv(char, char)	division
	<=	bool charle(char, char)	less or equal
	<	bool charlt(char, char)	less than
	=	bool chareq(char, char)	equality
	>=	bool charge(char, char)	greater or equal
	>	bool chargt(char, char)	greater than
char2	<>	bool char2ne(char2, char2)	inequality
	!~	bool char2regexne(char2, text)	A does not match regular expression B, case-sensitive
	!~*	bool char2icregexne(char2, text)	A does not match regular expression B, case-insensitive
	<=	bool char2le(char2, char2)	less or equal
	<	bool char2lt(char2, char2)	less than
	=	bool char2eq(char2, char2)	equality
	>=	bool char2ge(char2, char2)	greater or equal
	>	bool char2gt(char2, char2)	greater than
	~	bool char2regexeq(char2, text)	A matches regular expression B, case-sensitive.
	~*	bool char2icregexeq(char2, text)	A matches regular expression B, case-insensitive.
char4	<>	bool char4ne(char4, char4)	inequality
	!~	bool char4regexne(char4, text)	A does not match regular expression B, case-sensitive
	!~*	bool char4icregexne(char4, text)	A does not match regular expression B, case-insensitive
	<=	bool char4le(char4, char4)	less or equal
	<	bool char4lt(char4, char4)	less than
	=	bool char4eq(char4, char4)	equality
	>=	bool char4ge(char4, char4)	greater or equal

	>	bool char4gt(char4, char4)	greater than
	~	bool char4regexeq(char4, text)	A matches regular expression B, case-sensitive
	~*	bool char4icregexeq(char4, text)	A matches regular expression B, case-insensitive
char8	<>	bool char8ne(char8, char8)	inequality
	!~	bool char8regexne(char8, text)	A does not match regular expression B, case sensitive
	!~*	bool char8icregexne(char8, text)	A does not match regular expression B, case insensitive
	<=	bool char8le(char8, char8)	less or equal
	<	bool char8lt(char8, char8)	less than
	=	bool char8eq(char8, char8)	equality
	>=	bool char8ge(char8, char8)	greater or equal
	>	bool char8gt(char8, char8)	greater than
	~	bool char8regexeq(char8, text)	A matches regular expression B, case sensitive
	~*	bool char8icregexeq(char8, text)	A matches regular expression B, case insensitive
char16	<>	bool char16ne(char16, char16)	inequality
	!~	bool char16regexne(char16, text)	A does not match regular expression B, case sensitive
	!~*	bool char16icregexne(char16, text)	A does not match regular expression B, case insensitive
	<=	bool char16le(char16, char16)	less or equal
	<	bool char16lt(char16, char16)	less than
	=	bool char16eq(char16, char16)	equality
	>=	bool char16ge(char16, char16)	greater or equal
	>	bool char16gt(char16, char16)	greater than
	~	bool char16regexeq(char16, text)	A matches regular expression B, case sensitive
	~*	bool char16regexeq(char16, text)	A matches regular expression B, case insensitive
float4	<>	bool float4ne(float4, float4)	inequality
	*	float4 float4mul(float4, float4)	multiplication
	+	float4 float4pl(float4, float4)	addition
	–	float4 float4mi(float4, float4)	subtraction
	/	float4 float4div(float4, float4)	division
	<=	bool float4le(float4, float4)	less or equal
	<	bool float4lt(float4, float4)	less than
	=	bool float4eq(float4, float4)	equality
	>=	bool float4ge(float4, float4)	greater or equal
	>	bool float4gt(float4, float4)	greater than
float8	<>	bool float8ne(float8, float8)	inequality
	*	float8 float8mul(float8, float8)	multiplication
	+	float8 float8pl(float8, float8)	addition
	–	float8 float8mi(float8, float8)	subtraction
	/	float8 float8div(float8, float8)	division
	<=	bool float8le(float8, float8)	less or equal

	<	bool float8lt(float8, float8)	less than1
	=	bool float8eq(float8, float8)	equality
	>=	bool float8ge(float8, float8)	greater or equal
	>	bool float8gt(float8, float8)	greater than
	ˆ	float8 dpow(float8, float8)	exponentiation
int2	<>	bool int2ne(int2, int2)	inequality
	<>	int4 int24ne(int2, int4)	inequality
	%	int2 int2mod(int2, int2)	modulus
	%	int4 int24mod(int2, int4)	modulus
	*	int2 int2mul(int2, int2)	multiplication
	*	int4 int24mul(int2, int4)	multiplication
	+	int2 int2pl(int2, int2)	addition
	+	int4 int24pl(int2, int4)	addition
	−	int2 int2mi(int2, int2)	subtraction
	−	int4 int24mi(int2, int4)	subtraction
	/	int2 int2div(int2, int2)	division
	/	int4 int24div(int2, int4)	division
	<=	bool int2le(int2, int2)	less or equal
	<=	bool int24le(int2, int4)	less or equal
	<	bool int2lt(int2, int2)	less than
	<	bool int24lt(int2, int4)	less than
	=	bool int2eq(int2, int2)	equality
	=	bool int24eq(int2, int4)	equality
	>=	bool int2ge(int2, int2)	greater or equal
	>=	bool int24ge(int2, int4)	greater or equal
	>	bool int2gt(int2, int2)	greater than
	>	bool int24gt(int2, int4)	greater than
int4	<>	bool int4ne(int4, int4)	inequality
	<>	bool int42ne(int4, int2)	inequality
	%	int4 int42mod(int4, int2)	modulus
	%	int4 int4mod(int4, int4)	modulus
	*	int4 int42mul(int4, int2)	multiplication
	*	int4 int4mul(int4, int4)	multiplication
	+	int4 int42pl(int4, int2)	addition
	+	int4 int4pl(int4, int4)	addition
	−	int4 int42mi(int4, int2)	subtraction
	−	int4 int4mi(int4, int4)	subtraction
	/	int4 int42div(int4, int2)	division
	/	int4 int4div(int4, int4)	division
	<=	bool int4le(int4, int4)	less or equal
	<=	bool int42le(int4, int2)	less or equal
	<	bool int4lt(int4, int4)	less than
	<	bool int42lt(int4, int2)	less than
	=	bool int4eq(int4, int4)	equality
	=	bool int42eq(int4, int2)	equality
	=	bool int4eqoid(int4, oid)	equality
	>=	bool int4ge(int4, int4)	greater or equal
	>=	bool int42ge(int4, int2)	greater or equal
	>	bool int4gt(int4, int4)	greater than

	>	bool int42lt(int4, int2)	less than
-			
name	<>	bool namene(name, name)	inequality
	!~	bool nameregexne(name, text)	A does not match regular expression B, case sensitive
	!~*	bool nameicregexne(name, text)	A does not match regular expression B, case insensitive
	<=	bool namele(name, name)	less or equal
	<	bool namelt(name, name)	less than
	=	bool nameeq(name, name)	equality
	>=	bool namege(name, name)	greater or equal
	>	bool namegt(name, name)	greater than
	~	bool nameregexeq(name, text)	A matches regular expression B, case sensitive
	~*	bool nameregexeq(name, text)	A matches regular expression B, case insensitive
oid	<>	bool oidne(oid, oid)	inequality
	<>	bool oidne(oid, regproc)	inequality
	<=	bool oidle(oid, oid)	less or equal
	<	bool oidlt(oid, oid)	less than
	=	bool oideq(oid, oid)	equality
	=	bool oideqint4(oid, int4)	equality
	=	bool oideq(oid, regproc)	equality
	>=	bool oidge(oid, oid)	greater or equal
	>	bool oidgt(oid, oid)	greater than
oidname	<>	bool oidnamene(oidname, oidname)	inequality
	<	bool oidnamelt(oidname, oidname)	less than
	<=	bool oidnamele(oidname, oidname)	less or equal
	=	bool oidnameeq(oidname, oidname)	equality
	>	bool oidnamegt(oidname, oidname)	greater than
	>=	bool oidnamege(oidname, oidname)	greater or equal
oidint2	<>	bool oidint2ne(oidint2, oidint2)	inequality
	<	bool oidint2lt(oidint2, oidint2)	less than
	<=	bool oidint2le(oidint2, oidint2)	less or equal
	=	bool oidint2eq(oidint2, oidint2)	equality
	>	bool oidint2gt(oidint2, oidint2)	greater than
	>=	bool oidint2ge(oidint2, oidint2)	greater or equal
oidint4	<>	bool oidint4ne(oidint4, oidint4)	inequality
	<	bool oidint4lt(oidint4, oidint4)	less than
	<=	bool oidint4le(oidint4, oidint4)	less or equal
	=	bool oidint4eq(oidint4, oidint4)	equality
	>	bool oidint4gt(oidint4, oidint4)	greater than
	>=	bool oidint4ge(oidint4, oidint4)	greater or equal
point	!<	bool point_left(point, point)	A is left of B
	!>	bool point_right(point, point)	A is right of B
	!^	bool point_above(point, point)	A is above B
	!|	bool point_below(point, point)	A is below B
	=|=	bool point_eq(point, point)	equality
	===>	bool on_pb(point, box)	point inside box

	===‘	bool on_ppath(point, path)	point on path
	<===>	int4 pointdist(point, point)	distance between points
polygon	&&	bool poly_overlap(polygon, polygon)	polygons overlap
	&<	bool poly_overleft(polygon, polygon)	A overlaps B but does not extend to right of B
	&>	bool poly_overright(polygon, polygon)	A overlaps B but does not extend to left of B
	<<	bool poly_left(polygon, polygon)	A is left of B
	>>	bool poly_right(polygon, polygon)	A is right of B
	@	bool poly_contained(polygon, polygon)	A is contained by B
	~=	bool poly_same(polygon, polygon)	equality
	~	bool poly_contain(polygon, polygon)	A contains B
reltime	<>	bool reltimene(reltime, reltime)	inequality
	<=	bool reltimele(reltime, reltime)	less or equal
	<	bool reltimelt(reltime, reltime)	less than
	=	bool reltimeeq(reltime, reltime)	equality
	>=	bool reltimege(reltime, reltime)	greater or equal
	>	bool reltimegt(reltime, reltime)	greater than
text	<>	bool textne(text, text)	inequality
	!~	bool textregexne(text, text)	A does not contain the regular expression B, case sensitive
	!~*	bool texticregexne(text, text)	A does not contain the regular expression B, case insensitive
	<=	bool text_le(text, text)	less or equal
	<	bool text_lt(text, text)	less than
	=	bool texteq(text, text)	equality
	>=	bool text_ge(text, text)	greater or equal
	>	bool text_gt(text, text)	greater than
	~	bool textregexeq(text, text)	A contains the regular expression B. case sensitive
	~*	bool texticregexeq(text, text)	A contains the regular expression B. case insensitive
tinterval	#<>	bool intervallenne(tinterval, reltime)	interval length not equal to reltime.
	#<=	bool intervallenle(tinterval, reltime)	interval length less or equal reltime
	#<	bool intervallenlt(tinterval, reltime)	interval length less than reltime
	#=	bool intervalleneq(tinterval, reltime)	interval length equal to reltime
	#>=	bool intervallenge(tinterval, reltime)	interval length greater or equal reltime
	#>	bool intervallengt(tinterval, reltime)	interval length greater than reltime
	&&	bool intervalov(tinterval, tinterval)	intervals overlap
	<<	bool intervalct(tinterval, tinterval)	A contains B
	=	bool intervaleq(tinterval, tinterval)	equality
	<>	tinterval mktinterval(abstime, abstime)	interval bounded by two abstimes

UNARY OPERATORS

The tables below give right and left unary operators. Left unary operators have the operator precede the operand; right unary operators have the operator follow the operand.

Right Unary Operators

Type	Operator	POSTGRES Function Prototype	Operation
float8	%	float8 dround(float8)	round to nearest integer

Left Unary Operators

<table>
<tr><th>Type</th><th>Operator</th><th>POSTGRES Function Prototype</th><th>Operation</th></tr>
<tr><td>box</td><td>@@</td><td>point box_center(box)</td><td>center of box</td></tr>
<tr><td>float4</td><td>@</td><td>float4 float4abs(float4)</td><td>absolute value</td></tr>
<tr><td>float8</td><td>@</td><td>float8 float8abs(float8)</td><td>absolute value</td></tr>
<tr><td></td><td>%</td><td>float8 dtrunc(float8)</td><td>truncate to integer</td></tr>
<tr><td></td><td>|/</td><td>float8 dsqrt(float8)</td><td>square root</td></tr>
<tr><td></td><td>||/</td><td>float8 dcbrt(float8)</td><td>cube root</td></tr>
<tr><td></td><td>:</td><td>float8 dexp(float8)</td><td>exponential function</td></tr>
<tr><td></td><td>;</td><td>float8 dlog1(float8)</td><td>natural logarithm</td></tr>
<tr><td>tinterval</td><td>|</td><td>abstime intervalstart(tinterval)</td><td>start of interval</td></tr>
</table>

AGGREGATE FUNCTIONS

The table below gives the aggregate functions that are registered in the system catalogs. They operate on int2, int4, float4, and float8 types.

Name	Operation
arg	average
sum	sum
max	maximum
min	minimum
count	count

SEE ALSO

For examples on specifying literals of built-in types, see *SQL(commands).*

BUGS

Although most of the input and output functions correponding to the base types (e.g., integers and floating point numbers) do some error-checking, none of them are particularly rigorous about it. More importantly, almost none of the operators and functions (e.g., addition and multiplication) perform any error-checking at all. Consequently, many of the numeric operations will (for example) silently underflow or overflow.

Some of the input and output functions are not invertible. That is, the result of an output function may lose precision when compared to the original input.

SECTION 4 — SQL COMMANDS (COMMANDS)

DESCRIPTION

The following is a description of the general syntax of SQL. Individual SQL statements and commands are treated separately in the document; this section describes the syntactic classes from which the constituent parts of SQL statements are drawn.

Comments

A *comment* is an arbitrary sequence of characters following double dashes up to the end of the line e.g:

```
-- This is a comment
```

Names

Names in SQL are sequences of not more than NAMEDATALEN alphanumeric characters, starting with an alphabetic character. By default, NAMEDATALEN is set to 16, but at the time the system is built, NAMEDATALEN can be changed by changing the #ifdef in src/backend/include/postgres.h. Underscore ("_") is considered an alphabetic character.

Keywords

The following identifiers are reserved for use as *keywords* and may not be used otherwise:

In addition, all POSTGRES95 classes have several predefined attributes used by the system.

Constants

There are six types of *constants* for use in SQL. They are described below.

String Constants

Strings in SQL are arbitrary sequences of ASCII characters bounded by single quotes (' '). Uppercase alphabetics within strings are accepted literally. Non-printing characters may be embedded within strings by prepending them with a backslash, e.g., '\n'. Also, in order to embed quotes within strings, it is necessary to prefix them with '\' . The same convention applies to '\' itself. Because of the limitations on instance sizes, string constants are currently limited to a length of a little less than 8192 bytes. Larger objects may be created using the POSTGRES95 Large Object interface.

Integer Constants

Integer constants in SQL are collection of ASCII digits with no decimal point. Legal values range from –2147483647 to +2147483647. This will vary depending on the operating system and host machine.

Floating Point Constants

Floating point constants consist of an integer part, a decimal point, and a fraction part or scientific notation of the following format:

```
{<dig>} .{<dig>} [e [+-] {<dig>}]
```

Where <dig> is a digit. You must include at least one <dig> after the period and after the [+-] if you use those options. An exponent with a missing mantissa has a mantissa of 1 inserted. There may be no extra characters embedded in the string. Floating point constaints are of type float4.

Constants of POSTGRES95 User-Defined Types

A constant of an *arbitrary* type can be entered using the notation:

or

```
CAST 'string' AS type-name
```

The value inside the string is passed to the input conversion routine for the type called type-name. The result is a constant of the indicated type. The explicit typecast may be omitted if there is no ambiguity as to the type the constant must be, in which case it is automatically coerced.

Array constants

Array constants are arrays of any POSTGRES95 type, including other arrays, string constants, etc. The general format of an array constant is the following:

```
{<val1><delim><val2><delim>}
```

Where *<delim>* is the delimiter for the type stored in the "pg_type" class. (For built-in types, this is the comma character, ",".) An example of an array constant is

```
{{1,2,3},{4,5,6},{7,8,9}}
```

This constant is a two-dimensional, 3 by 3 array consisting of three sub-arrays of integers.

Individual array elements can and should be placed between quotation marks whenever possible to avoid ambiguity problems with respect to leading white space.

Fields

A *field* is either an attribute of a given class or one of the following:

```
oid
tmin
tmax
xmin
xmax
cmin
cmax
```

Oid stands for the unique identifier of an instance which is added by POSTGRES95 to all instances automatically. Oids are not reused and are 32 bit quantities.

Tmin, tmax, xmin, cmin, xmax and *cmax* stand respectively for the time that the instance was inserted, the time the instance was deleted, the identity of the inserting transaction, the command identifier within the transaction, the identity of the deleting transaction and its associated deleting command. For further information on these fields consult [STON87]. Times are represented internally as instances of the "abstime" data type. Transaction identifiers are 32 bit quantities which are assigned sequentially starting at 512. Command identifiers are 16 bit objects; hence, it is an error to have more than 65535 SQL commands within one transaction.

Columns

A *column* is a construct of the form:

```
Instance-variable{.composite_field}.field `['number`]'
```

Instance-variable identifies a particular class and can be thought of as standing for the instances of that class. An instance variable is either a class name, a surrogate for a class defined by means of a *from* clause, or the keyword **new** or **current.** New and current can only appear in the action portion of a rule, while other instance variables can be used in any SQL statement. *Composite_field* is a field of of one of the POSTGRES95 composite types indicated in the *information*(commands) section, while successive composite fields address attributes in the class(s) to which the composite field evaluates. Lastly, *field* is a normal (base type) field in the class(s) last addressed. If *field* is of type array, then the optional *number* designator indicates a specific element in the array. If no number is indicated, then all array elements are returned.

Operators

Any built-in system, or user-defined operator may be used in SQL. For the list of built-in and system operators consult **introduction**(built-ins). For a list of user-defined operators consult your system administrator or run a query on the pg_operator class. Parentheses may be used for arbitrary grouping of operators.

Expressions (a_expr)

An *expression* is one of the following:

```
( a_expr )
constant
attribute
a_expr binary_operator a_expr
a_expr right_unary_operator
left_unary_operator a_expr
parameter
functional expressions
aggregate expressions
```

We have already discussed constants and attributes. The two kinds of operator expressions indicate respectively binary and left_unary expressions. The following sections discuss the remaining options.

Parameters

A *parameter* is used to indicate a parameter in a SQL function. Typically this is used in SQL function definition statement. The form of a parameter is:

```
´$’ number
```

For example, consider the definition of a function, DEPT, as

```
create function DEPT (char16)
      returns dept
      as ’select * from
          dept where name=$1’
      language ’sql’
```

Functional Expressions

A *functional expression* is the name of a legal SQL function, followed by its argument list enclosed in parentheses, e.g.:

```
fn−name (a_expr{ , a_expr})
```

For example, the following computes the square root of an employee salary.

```
sqrt(emp.salary)
```

Aggregate Expression

An *aggregate expression* represents a simple aggregate (i.e., one that computes a single value) or an aggregate function (i.e., one that computes a set of values). The syntax is the following:

```
aggregate.name (attribute)
```

Here, *aggregate_name* must be a previously defined aggregate.

Target_list

A *target list* is a parenthesized, comma-separated list of one or more elements, each of which must be of the form:

```
a_expr[AS result_attname]
```

Here, result_attname is the name of the attribute to be created (or an already existing attribute name in the case of update statements.) If *result_attname* is not present, then *a_expr* must contain only one attribute name which is assumed to be the name of the result field. In POSTGRES95 default naming is only used if *a_expr* is an attribute.

Qualification

A *qualification* consists of any number of clauses connected by the logical operators:

```
not
and
or
```

A clause is an *a_expr* that evaluates to a Boolean over a set of instances.

From List

The *from list* is a comma-separated list of *from expressions*.

Each *from expression* is of the form:

```
[class_reference] instance_variable
      {, [class_ref] instance_variable...}
```

where *class_reference* is of the form

```
class_name [time_expression] [*]
```

The *from expression* defines one or more instance variables to range over the class indicated in *class_reference*. Adding a *time_expression* will indicate that a historical class is desired. One can also request the instance variable to range over all classes that are beneath the indicated class in the inheritance hierarchy by postpending the designator “*”.

Time Expressions

A *time expression* is in one of two forms:

```
['date']
['date-1', 'date-2']
```

The first case requires instances that are valid at the indicated time. The second case requires instances that are valid at some time within the date range specified. If no time expression is indicated, the default is "now".

In each case, the date is a character string of the form

```
[MON-FRI] 'MMM DD [HH:MM:SS] YYYY' [Timezone]
```

where MMM is the month (Jan – Dec), DD is a legal day number in the specified month, HH:MM:SS is an optional time in that day (24-hour clock), and YYYY is the year. If the time of day HH:MM:SS is not specified, it defaults to midnight at the start of the specified day. As of Version 3.0, times are no longer read and written using Greenwich Mean Time; the input and output routines default to the local time zone.

For example,

```
['Jan 1 1990']
['Mar 3 00:00:00 1980', 'Mar 3 23:59:59 1981r']
```

are valid time specifications.

Note that this syntax is slightly different than that used by the time-range type.

SEE ALSO

insert(commands), delete(commands), execute(commands), update(commands), select(commands), monitor(unix).

NAME

abort — abort the current transaction

SYNOPSIS

abort [transaction]

DESCRIPTION

This command aborts the current transaction and causes all the updates made by the transaction to be discarded. *abort* is functionally equivalent to *rollback*.

SEE ALSO

begin(commands), end(commands), rollback(commands).

NAME

alter table — add attributes to a class

SYNOPSIS

alter table classname [*]
add column attname type

DESCRIPTION

The **alter table** command causes a new attribute to be added to an existing class, *classname*. The new attributes and their types are specified in the same style and with the the same restrictions as in *create*table*(commands).*

In order to add an attribute to each class in an entire inheritance hierarchy, use the *classname* of the superclass and append a "*". (By default, the attribute will not be added to any of the subclasses.) This should **always** be done when adding an attribute to a superclass. If it is not, queries on the inheritance hierarchy such as

```
select * from super* s
```

will not work because the subclasses will be missing an attribute found in the superclass.

For efficiency reasons, default values for added attributes are not placed in existing instances of a class. That is, existing instances will have NULL values in the new attributes. If non-NULL values are desired, a subsequent *update*(commands) query should be run.

You must own the class in order to change its schema.

EXAMPLE

```
--
-- add the date of hire to the emp class
--
alter table emp add column hiredate abstime


--
-- add a health-care number to all persons
-- (including employees, students, ...)
--
alter table person * add column health_care_id int4
```

SEE ALSO

create table (commands), update (commands).

NAME

begin — begins a transaction

SYNOPSIS

begin [transaction|work]

DESCRIPTION

This command begins a user transaction which POSTGRES95 will guarantee is serializable with respect to all concurrently executing transactions. POSTGRES95 uses two-phase locking to perform this task. If the transaction is committed, POSTGRES95 will ensure that all updates are done or none of them are done. Transactions have the standard ACID (atomic, consistent, isolatable, and durable) property.

SEE ALSO

abort(commands), end(commands).

NAME

close — close a cursor

SYNOPSIS

close [cursor_name]

DESCRIPTION

Close frees the resources associated with a cursor, *cursor_name.* After this cursor is closed, no subsequent operations are allowed on it. A cursor should be closed when it is no longer needed. If *cursor_name.* is not specified, then the blank cursor is closed.

EXAMPLE

```
/*
 * close the cursor FOO
 */
close FOO
```

SEE ALSO

fetch(commands), select(commands).

NAME

commit — commit the current transaction

SYNOPSIS

commit [transaction|work]

DESCRIPTION

This commands commits the current transaction. All changes made by the transaction become visible to others and are guaranteed to be durable if a crash occurs. *commit* is functionally equivalent to the *end* command

SEE ALSO

abort(commands), begin(commands), end(commands), rollback(commands).

NAME

copy — copy data to or from a class from or to a UNIX file.

SYNOPSIS

copy [**binary**] [**nonulls**] classname
  to|**from** "filename"|**stdin**|**stdout**
  [**USING DELIMITERS** delim]

DESCRIPTION

Copy moves data between POSTGRES95 classes and standard UNIX files. The keyword **binary** changes the behavior of field formatting, as described below. *Classname* is the name of an existing class. *Filename* is the UNIX pathname of the file. In place of a filename, the keywords **stdin** and **stdout** can be used so that input to **copy** can be written by a LIBPQ application and output from the **copy** command can be read by a LIBPQ application. The **binary** keyword will force all data to be stored/read as binary objects rather than as ASCII text. It is somewhat faster than the normal **copy** command, but is not generally portable, and the files generated are somewhat larger, although this factor is highly dependent on the data itself.

By default, **copy** uses a tab (\t) character as a delimiter. The delimiter may also be changed to any other single-character with the use of **USING DELIMITERS** Characters in data fields which happen to match the delimiter character will be quoted.

You must have read access on any class whose values are read by the **copy** command, and either write or append access to a class to which values are being appended by the **copy** command.

FORMAT OF OUTPUT FILES

ASCII COPY FORMAT

When **copy** is used without the **binary** keyword, the file generated will have each instance on a line, with each attribute separated by the delimiter character. Embedded delimiter characters will be preceeded by a backslash character (\). The attribute values themselves are strings generated by the output function associated with each attribute type. The output function for a type should not try to generate the backslash character; this will be handled by **copy** itself.

Note that on input to **copy**, backslashes are considered to be special control characters, and should be doubled if you want to embed a backslash, i.e., the string “12\\19\88” will be converted by **copy** to “12\1988”. The actual format for each instance is

  <attr1><tab><attr2><tab>...<tab><attrn><newline>

If **copy** is sending its output to standard output instead of a file, it will send a period (.) followed immediately by a newline, on a line by themselves, when it is done. Similarly, if **copy** is reading from standard input, it will expect a period (.) followed by a newline, as the first two characters on a line, to denote end-of-file. However, **copy** will terminate (followed by the backend itself) if a true EOF is encountered.

NULL attributes are handled simply as null strings, that is, consecutive tabs in the input file denote a **NULL** attribute.

BINARY COPY FORMAT

In the case of **copy binary**, the first four bytes in the file will be the number of instances in the file. If this number is *zero,* the **copy binary** command will read until end of file is encountered. Otherwise, it will stop reading when this number of instances has been read. Remaining data in the file will be ignored.

The format for each instance in the file is as follows. Note that this format must be followed **EXACTLY**. Unsigned four-byte integer quantities are called uint32 in the below description.

  uint32 totallength (not including itself),
  uint32 number of null attributes
  [uint32 attribute number of first null attribute
  ...

 COPY(COMMANDS)

uint32 attribute number of nth null attribute],
<data>

ALIGNMENT OF BINARY DATA

On Sun-3s, 2-byte attributes are aligned on two-byte boundaries, and all larger attributes are aligned on four-byte boundaries. Character attributes are aligned on single-byte boundaries. On other machines, all attributes larger than 1 byte are aligned on four-byte boundaries. Note that variable length attributes are preceded by the attribute's length; arrays are simply contiguous streams of the array element type.

SEE ALSO

insert(commands), create table(commands), vacuum(commands), libpq.

BUGS

Files used as arguments to the **copy** command must reside on or be accessible to the the database server machine by being either on local disks or a networked file system.

Copy stops operation at the first error. This should not lead to problems in the event of a **copy from**, but the target relation will, of course, be partially modified in a **copy to**. The *vacuum*(commands) query should be used to clean up after a failed **copy**.

Because POSTGRES95 operates out of a different directory than the user's working directory at the time POSTGRES95 is invoked, the result of copying to a file "foo" (without additional path information) may yield unexpected results for the naive user. In this case, "foo" will wind up in $PGDATA/foo. In general, the full pathname should be used when specifying files to be copied.

Copy has virtually no error checking, and a malformed input file will likely cause the backend to crash. You should avoid using **copy** for input whenever possible.

NAME

create aggregate — define a new aggregate

SYNOPSIS

create aggregate agg-name [**as**]
([**sfunc1 =** state-transition-function-1
, **basetype =** data-type
, **stype1 =** sfunc1-return-type]
[**, sfunc2 =** state-transition-function-2
, **stype2 =** sfunc2-return-type]
[**, finalfunc =** final-function]
[**, initcond1 =** initial-condition-1]
[**, initcond2 =** initial-condition-2])

DESCRIPTION

An aggregate function can use up to three functions, two *state transition* functions, X1 and X2:

X1(internal-state1, next-data_item) ---> next-internal-state1
X2(internal-state2) ---> next-internal-state2

and a **final calculation** function, F:

F(internal-state1, internal-state2) ---> aggregate-value

These functions are required to have the following properties:

The arguments to state-transition-function-1 must be (stype1**,**basetype), and its return value must be stype1.

The argument and return value of state-transition-function-2 must be **stype2**.

The arguments to the final-calculation-function must be (stype1**,**stype2), and its return value must be a POSTGRES base type (not necessarily the same as basetype.

The final-calculation-function should be specified if and only if both state-transition functions are specified.

Note that it is possible to specify aggregate functions that have varying combinations of state and final functions. For example, the "count" aggregate requires **sfunc2** (an incrementing function) but not **sfunc1** or **finalfunc**, whereas the "sum" aggregate requires **sfunc1** (an addition function) but not **sfunc2** or **finalfunc** and the "average" aggregate requires both of the above state functions as well as a **finalfunc** (a division function) to produce its answer. In any case, at least one state function must be defined, and any **sfunc2** must have a corresponding **initcond2**.

Aggregates also require two initial conditions, one for each transition function. These are specified and stored in the database as fields of type *text*.

EXAMPLE

This *avg* aggregate consists of two state transition functions, a addition function and a incrementing function. These modify the internal state of the aggregate through a running sum and and the number of values seen so far. It accepts a new employee salary, increments the count, and adds the new salary to produce the next state. The state transition functions must be passed correct initialization values. The final calculation then divides the sum by the count to produce the final answer.

```
--
--Create an aggregate for int4 average
--
create aggregate avg (sfunc1 = int4add, basetype = int4,
     stype1 = int4, sfunc2 = int4inc, stype2 = int4,
     finalfunc = int4div, initcond1 = "0", initcond2 = "0")
```

SEE ALSO

create function(commands), remove aggregate(commands).

NAME

create function — define a new function

SYNOPSIS

create function function_name (
([type1 {, type-n}])
returns type-r
as {'/full/path/to/objectfile' | 'sql-queries'}
language {'c' 'sql' 'internal'}

DESCRIPTION

With this command, a POSTGRES95 user can register a function with POSTGRES95. Subsequently, this user is treated as the owner of the function.

When defining a function with arguments, the input data types, *type-1*, *type-2*, ..., *type-n*, and the return data type, *type-r* must be specified, along with the language, which may be *"c"* or *"sql"*. or *"internal"*. (The *arg is* clause may be left out if the function has no arguments, or alternatively the argument list may be left empty.) The input types may be base or complex types, or *opaque*. *Opaque* indicates that the function accepts arguments of an invalid type such as (char *). The output type may be specified as a base type, complex type, *setof* <*type*>, or *opaque*. The *setof* modifier indicates that the function will return a set of items, rather than a single item. The *as* clause of the command is treated differently for C and SQL functions, as explained below.

C FUNCTIONS

Functions written in C can be defined to POSTGRES95, which will dynamically load them into its address space. The loading happens either using *load*(commands) or automatically the first time the function is necessary for execution. Repeated execution of a function will cause negligible additional overhead, as the function will remain in a main memory cache.

Internal functions are functions written in C which have been statically linked into the postgres backend process. The **as** clause must still be specified when defining an internal function but the contents are ignored.

Writing C Functions

The body of a C function following **as** should be the **FULL PATH** of the object code (.o file) for the function, bracketed by quotation marks. (POSTGRES95 will not compile a function automatically — it must be compiled before it is used in a **define function** command.)

C functions with base type arguments can be written in a straightforward fashion. The C equivalents of built-in POSTGRES95 types are accessible in a C file if

```
.../src/backend/utils/builtins.h
```

is included as a header file. This can be achieved by having

```
#include <utils/builtins.h>
```

at the top of the C source file and by compiling all C files with the following include options:

```
-I.../src/backend
-I.../src/backend/port/<portname>
-I.../src/backend/obj
```

before any ".c" programs in the *cc* command line, e.g.:

```
cc -I.../src/backend \
   -I.../src/backend/port/<portname> \
   -I.../src/backend/obj \
   -c progname.c
```

where "..." is the path to the installed POSTGRES95 source tree and "<portname>" is the name of the port for which the source tree has been built.

The convention for passing arguments to and from the user's C functions is to use pass-by-value for data types that are 32 bits (4 bytes) or smaller, and pass-by-reference for data types that require more than 32 bits.

The following table gives the C type required for parameters in the C functions that will be loaded into POSTGRES95. The "Defined In" column gives the actual header file (in the

```
.../src/backend
```

directory) that the equivalent C type is defined. However, if you include "utils/builtins.h", these files will automatically be included.

Equivalent C Types for Built-In POSTGRES95 Types

Built-In Type	C Type	Defined In
abstime	AbsoluteTime	utils/nabstime.h
bool	bool	include/c.h
box	(BOX *)	utils/geo-decls.h
bytea	(bytea *)	include/postgres.h
char	char	N/A
char16	Char16 or (char16 *)	include/postgres.h
cid	CID	include/postgres.h
int2	int2	include/postgres.h
int28	(int28 *)	include/postgres.h
int4	int4	include/postgres.h
float4	float32 or (float4 *)	include/c.h or include/postgres.h
float8	float64 or (float8 *)	include/c.h or include/postgres.h
lseg	(LSEG *)	include/geo-decls.h
name	(Name)	include/postgres.h
oid	oid	include/postgres.h
oid8	(oid8 *)	include/postgres.h
path	(PATH *)	utils/geo-decls.h
point	(POINT *)	utils/geo-decls.h
regproc	regproc or REGPROC	include/postgres.h
reltime	RelativeTime	utils/nabstime.h
text	(text *)	include/postgres.h
tid	ItemPointer	storage/itemptr.h
tinterval	TimeInterval	utils/nabstime.h
uint2	uint16	include/c.h
uint4	uint32	include/c.h
xid	(XID *)	include/postgres.h

Complex arguments to C functions are passed into the C function as a special C type, TUPLE, defined in

```
.../src/libpq/libpq-fe.h.
```

Given a variable *t* of this type, the C function may extract attributes from the function using the function call:

```
GetAttributeByName(t, "fieldname", &isnull)
```

where *isnull* is a pointer to a *bool*, which the function sets to *true* if the field is null. The result of this function should be cast appropriately as shown in the examples below.

Compiling Dynamically-Loaded C Functions

Different operating systems require different procedures for compiling C source files so that POSTGRES95 can load them dynamically. This section discusses the required compiler and loader options on each system.

Under Linux ELF, object files can be generated by specifing the compiler flag -fpic.

Under Ultrix, all object files that POSTGRES95 is expected to load dynamically must be compiled using */bin/cc* with the "-G 0" option turned on. The object file name in the *as* clause should end in ".o".

Under HP-UX, DEC OSF/1, AIX and SunOS 4, all object files must be turned into *shared libraries* using the operating system's native object file loader, *ld*(1).

Under HP-UX, an object file must be compiled using the native HP-UX C compiler, */bin/cc*, with both the "+z" and "+u" flags turned on. The first flag turns the object file into "position-independent code" (PIC); the second flag removes some alignment restrictions that the PA-RISC architecture normally enforces. The object file must then be turned into a shared library using the HP-UX loader, */bin/ld*. The command lines to compile a C source file, "foo.c", look like:

```
cc <other flags> +z +u -c foo.c
ld <other flags> -b -o foo.sl foo.o
```

The object file name in the **as** clause should end in ".sl".

An extra step is required under versions of HP-UX prior to 9.00. If the POSTGRES95 header file

```
include/c.h
```

is not included in the source file, then the following line must also be added at the top of every source file:

```
#pragma HP_ALIGN HPUX_NATURAL_S500
```

However, this line must not appear in programs compiled under HP-UX 9.00 or later.

Under DEC OSF/1, an object file must be compiled and then turned into a shared library using the OSF/1 loader, */bin/ld*. In this case, the command lines look like:

```
cc <other flags> -c foo.c
ld <other flags> -shared -expect_unresolved '*' -o foo.so foo.o
```

The object file name in the **as** clause should end in ".so".

Under SunOS 4, an object file must be compiled and then turned into a shared library using the SunOS 4 loader, */bin/ld*. The command lines look like:

```
cc <other flags> -PIC -c foo.c
ld <other flags> -dc -dp -Bdynamic -o foo.so foo.o
```

The object file name in the **as** clause should end in “.so”.

Under AIX, object files are compiled normally but building the shared library requires a couple of steps. First, create the object file:

```
cc <other flags> -c foo.c
```

You must then create a symbol “exports” file for the object file:

```
mkldexport foo.o `pwd` > foo.exp
```

Finally, you can create the shared library:

```
ld <other flags> -H512 -T512 -o foo.so -e _nostart \
   -bI:.../lib/postgres.exp -bE:foo.exp foo.o \
   -lm -lc 2>/dev/null
```

You should look at the POSTGRES95 User’s Manual for an explanation of this procedure.

SQL FUNCTIONS

SQL functions execute an arbitrary list of SQL queries, returning the results of the last query in the list. SQL functions in general return sets. If their returntype is not specified as a *setof*, then an arbitrary element of the last query’s result will be returned.

The body of a SQL function following **as** should be a list of queries separated by whitespace characters and bracketed within quotation marks. Note that quotation marks used in the queries must be escaped, by preceding them with two backslashes (i.e. \\").

Arguments to the SQL function may be referenced in the queries using a $n syntax: $1 refers to the first argument, $2 to the second, and so on. If an argument is complex, then a “dot” notation may be used to access attributes of the argument (e.g. “$1.emp”), or to invoke functions via a nested-dot syntax.

EXAMPLES: C Functions

The following command defines a C function, overpaid, of two basetype arguments.

```
create function overpaid (float8, int4) returns bool
         as '/usr/postgres/src/adt/overpaid.o'
         language 'c'
```

The C file "overpaid.c" might look something like:

```
#include <utils/builtins.h>

bool overpaid(salary, age)
        float8 *salary;
        int4    age;
{
        if (*salary > 200000.00)
                return(TRUE);
        if ((age < 30) & (*salary > 100000.00))
```

```
                return(TRUE);
        return(FALSE);
}
```

The overpaid function can be used in a query, e.g:

```
select name from EMP where overpaid(salary, age)
```

One can also write this as a function of a single argument of type EMP:

```
create function overpaid_2 (EMP)
         returns bool
         as '/usr/postgres/src/adt/overpaid_2.o'
         language 'c'
```

The following query is now accepted:

```
select name from EMP where overpaid_2(EMP)
```

In this case, in the body of the overpaid_2 function, the fields in the EMP record must be extracted. The C file "overpaid_2.c" might look something like:

```
#include <utils/builtins.h>
#include <libpq-fe.h>

bool overpaid_2(t)
TUPLE t;
{
    float8 *salary;
    int4    age;
    bool    salnull, agenull;

    salary = (float8 *)GetAttributeByName(t, "salary",
                                          &salnull);
    age = (int4)GetAttributeByName(t, "age", &agenull);
    if (!salnull && *salary > 200000.00)
        return(TRUE);
    if (!agenull && (age<30) && (*salary > 100000.00))
        return(TRUE);
    return(FALSE)
}
```

EXAMPLES: SQL Functions

To illustrate a simple SQL function, consider the following, which might be used to debit a bank account:

```
create function TP1 (int4, float8) returns int4
         as 'update BANK set balance = BANK.balance - $2
                      where BANK.acctountno = $1
             select(x = 1)'
             language 'sql'
```

A user could execute this function to debit account 17 by $100.00 as follows:

```
select (x = TP1( 17,100.0))
```

The following more interesting examples take a single argument of type EMP, and retrieve multiple results:

```
select function hobbies (EMP) returns set of HOBBIES
          as 'select (HOBBIES.all) from HOBBIES
                              where $1.name = HOBBIES.person'
          language 'sql'
```

SEE ALSO

information(unix), load(commands), drop function(commands).

NOTES

Name Space Conflicts

More than one function may be defined with the same name, as long as the arguments they take are different. In other words, function names can be *overloaded*. A function may also have the same name as an attribute. In the case that there is an ambiguity between a function on a complex type and an attribute of the complex type, the attribute will always be used.

RESTRICTIONS

The name of the C function must be a legal C function name, and the name of the function in C code must be exactly the same as the name used in **create function**. There is a subtle implication of this restriction: while the dynamic loading routines in most operating systems are more than happy to allow you to load any number of shared libraries that contain conflicting (identically-named) function names, they may in fact botch the load in interesting ways. For example, if you define a dynamically-loaded function that happens to have the same name as a function built into POSTGRES95, the DEC OSF/1 dynamic loader causes POSTGRES95 to call the function within itself rather than allowing POSTGRES95 to call your function. Hence, if you want your function to be used on different architectures, we recommend that you do not overload C function names.

There is a clever trick to get around the problem just described. Since there is no problem overloading SQL functions, you can define a set of C functions with different names and then define a set of identically-named SQL function wrappers that take the appropriate argument types and call the matching C function.

opaque cannot be given as an argument to a SQL function.

BUGS

C functions cannot return a set of values.

NAME

create index — construct a secondary index

SYNOPSIS

create index index-name
  on classname **using** am-name
  (attname type_class **)**

create index index-name
  on classname **using** am-name
  (funcname **(** attname−1 { , attname−i } **)** type_class **)**

DESCRIPTION

This command constructs an index called *index-name.*

Am-name is the name of the access method which is used for the index.

In the first syntax shown above, the key field for the index is specified as an attribute name and an associated *operator class*. An operator class is used to specify the operators to be used for a particular index. For example, a btree index on four-byte integers would use the *int4_ops* class; this operator class includes comparison functions for four-byte integers.

In the second syntax shown above, an index can be defined on the result of a user-defined function *funcname* applied to one or more attributes of a single class. These *functional indices* are primarily useful in two situations. First, functional indices can be used to simulate multikey indices. That is, the user can define a new base type (a simple combination of, say, "oid" and "int2") and the associated functions and operators on this new type such that the access method can use it. Once this has been done, the standard techniques for interfacing new types to access methods (described in the POSTGRES95 user manual) can be applied. Second, functional indices can be used to obtain fast access to data based on operators that would normally require some transformation to be applied to the base data. For example, say you have an attribute in class "myclass" called "pt" that consists of a 2D point type. Now, suppose that you would like to index this attribute but you only have index operator classes for 2D polygon types. You can define an index on the point attribute using a function that you write (call it "point_to_polygon") and your existing polygon operator class; after that, queries using existing polygon operators that reference "point_to_polygon(myclass.pt)" on one side will use the precomputed polygons stored in the functional index instead of computing a polygon for each and every instance in "myclass" and then comparing it to the value on the other side of the operator. Obviously, the decision to build a functional index represents a tradeoff between space (for the index) and execution time.

POSTGRES95 provides btree, rtree and hash access methods for secondary indices. The btree access method is an implementation of the Lehman-Yao high-concurrency btrees. The rtree access method implements standard rtrees using Guttman's quadratic split algorithm. The hash access method is an implementation of Litwin's linear hashing. We mention the algorithms used solely to indicate that all of these access methods are fully dynamic and do not have to be optimized periodically (as is the case with, for example, static hash access methods).

The operator classes defined on btrees are

```
int2_ops     char2_ops    oidint2_ops
int4_ops     char4_ops    oidint4_ops
int24_ops    char8_ops    oidname_ops
int42_ops    char16_ops   abstime_ops
float4_ops   name_ops
float8_ops   oid_ops
char_ops     text_ops
```

The *int24_ops* operator class is useful for constructing indices on int2 data, and doing comparisons against int4 data in query qualifications. Similarly, *int42_ops* support indices on int4 data that is to be compared against int2 data in queries.

The operator classes *oidint2_ops*, *oidint4_ops*, and *oidchar16_ops* represent the use of *functional indices* to simulate multi-key indices.

The POSTGRES95 query optimizer will consider using btree indices in a scan whenever an indexed attribute is involved in a comparison using one of

```
<     <=     =     >=     >
```

The operator classes defined on rtrees are

```
box_ops
bigbox_ops
poly_ops
```

Both box classes support indices on the “box” datatype in POSTGRES95. The difference between them is that *bigbox_ops* scales box coordinates down, to avoid floating point exceptions from doing multiplication, addition, and subtraction on very large floating-point coordinates. If the field on which your rectangles lie is about 20,000 units square or larger, you should use *bigbox_ops*. The *poly_ops* operator class supports rtree indices on “polygon” data.

The POSTGRES95 query optimizer will consider using an rtree index whenever an indexed attribute is involved in a comparison using one of

```
<<     &<     &>     >>     @     ~=     &&
```

The operator classes defined on the hash access method are

```
char_ops     int2_ops
char2_ops    int4_ops
char4_ops    float4_ops
char8_ops    float8_ops
char16_ops   oid_ops
name_ops     text_ops
```

The POSTGRES95 query optimizer will consider using a hash index whenever an indexed attribute is involved in a comparison using the

```
=
```

operator.

EXAMPLES

```
--
--Create a btree index on the emp class using the age attribute.
--
create index empindex on emp using btree (age int4_ops)
```

```
--
--Create a btree index on employee name.
--
create index empname
      on emp using btree (name char16_ops)


--
--Create an rtree index on the bounding rectangle of cities.
--
create index cityrect
      on city using rtree (boundbox box_ops)


--
--Create a rtree index on a point attribute such that we
--can efficiently use box operators on the result of the
--conversion function.  Such a qualification might look
--like "where point2box(points.pointloc) = boxes.box".
--
create index pointloc
      on points using rtree (point2box(location) box_ops)
```

NAME

create operator — define a new user operator

SYNOPSIS

create operator operator_name
([**leftarg =** type-1]
[**, rightarg =** type-2]
, **procedure =** func_name
[**, commutator =** com_op]
[**, negator =** neg_op]
[**, restrict =** res_proc]
[**, hashes**]
[**, join =** join_proc]
[**, sort =** sor_op1 {**,** sor_op2 }]
)

DESCRIPTION

This command defines a new user operator, *operator_name*. The user who defines an operator becomes its owner.

The *operator_name* is a sequence of up to sixteen punctuation characters. The following characters are valid for single-character operator names:

```
~ ! @ # % ^ & ` ?
```

If the operator name is more than one character long, it may consist of any combination of the above characters or the following additional characters:

```
| $ : + - * / < > =
```

At least one of *leftarg* and *rightarg* must be defined. For binary operators, both should be defined. For right unary operators, only *arg1* should be defined, while for left unary operators only *arg2* should be defined.

The name of the operator, *operator_name*, can be composed of symbols only. Also, the *func_name* procedure must have been previously defined using *create function*(commands) and must have one or two arguments.

The commutator operator is present so that POSTGRES95 can reverse the order of the operands if it wishes. For example, the operator area-less-than, >>>, would have a commutator operator, area-greater-than, <<<. Suppose that an operator, area-equal, ===, exists, as well as an area not equal, !==. Hence, the query optimizer could freely convert:

```
"0,0,1,1"::box >>> MYBOXES.description
```

to

```
MYBOXES.description <<< "0,0,1,1"::box
```

This allows the execution code to always use the latter representation and simplifies the query optimizer somewhat.

The negator operator allows the query optimizer to convert

```
not MYBOXES.description === "0,0,1,1"::box
```

to

```
MYBOXES.description !== "0,0,1,1"::box
```

If a commutator operator name is supplied, POSTGRES95 searches for it in the catalog. If it is found and it does not yet have a commutator itself, then the commutator's entry is updated to have the current (new) operator as its commutator. This applies to the negator, as well.

This is to allow the definition of two operators that are the commutators or the negators of each other. The first operator should be defined without a commutator or negator (as appropriate). When the second operator is defined, name the first as the commutator or negator. The first will be updated as a side effect.

The next two specifications are present to support the query optimizer in performing joins. POSTGRES95 can always evaluate a join (i.e., processing a clause with two tuple variables separated by an operator that returns a boolean) by iterative substitution [WONG76]. In addition, POSTGRES95 is planning on implementing a hash-join algorithm along the lines of [SHAP86]; however, it must know whether this strategy is applicable. For example, a hash-join algorithm is usable for a clause of the form:

```
MYBOXES.description === MYBOXES2.description
```

but not for a clause of the form:

```
MYBOXES.description <<< MYBOXES2.description.
```

The **hashes** flag gives the needed information to the query optimizer concerning whether a hash join strategy is usable for the operator in question.

Similarly, the two sort operators indicate to the query optimizer whether merge-sort is a usable join strategy and what operators should be used to sort the two operand classes. For the === clause above, the optimizer must sort both relations using the operator, <<<. On the other hand, merge-sort is not usable with the clause:

```
MYBOXES.description <<< MYBOXES2.description
```

If other join strategies are found to be practical, POSTGRES95 will change the optimizer and run-time system to use them and will require additional specification when an operator is defined. Fortunately, the research community invents new join strategies infrequently, and the added generality of user-defined join strategies was not felt to be worth the complexity involved.

The last two pieces of the specification are present so the query optimizer can estimate result sizes. If a clause of the form:

```
MYBOXES.description <<< "0,0,1,1"::box
```

is present in the qualification, then POSTGRES95 may have to estimate the fraction of the instances in MYBOXES that satisfy the clause. The function res_proc must be a registered function (meaning it is already defined using *define function*(commands)) which accepts one argument of the correct data type and returns a floating point number. The query optimizer simply calls this function, passing the parameter

```
"0,0,1,1"
```

and multiplies the result by the relation size to get the desired expected number of instances.

Similarly, when the operands of the operator both contain instance variables, the query optimizer must estimate the size of the resulting join. The function join_proc will return another floating point number which will be multiplied by the cardinalities of the two classes involved to compute the desired expected result size.

The difference between the function

```
my_procedure_1 (MYBOXES.description, "0,0,1,1"::box)
```

and the operator

```
MYBOXES.description === "0,0,1,1"::box
```

is that POSTGRES95 attempts to optimize operators and can decide to use an index to restrict the search space when operators are involved. However, there is no attempt to optimize functions, and they are performed by brute force. Moreover, functions can have any number of arguments while operators are restricted to one or two.

EXAMPLE

```
--
--The following command defines a new operator,
--area-equality, for the BOX data type.
--
create operator === (
      leftarg = box,
      rightarg = box,
      procedure = area_equal_procedure,
      commutator = ===,
      negator = !==,
      restrict = area_restriction_procedure,
      hashes,
      join = area-join-procedure,
      sort = <<<, <<<)
```

SEE ALSO

create function(commands), drop operator(commands).

BUGS

Operator names cannot be composed of alphabetic characters in POSTGRES95.

If an operator is defined before its commuting operator has been defined (a case specifically warned against above), a dummy operator with invalid fields will be placed in the system catalogs. This may interfere with the definition of later operators.

NAME

create rule — define a new rule

SYNOPSIS

create rule rule_name
  as on event
   to object [**where** clause]
  do [**instead**]
  [action | nothing | **[**actions...**]**]

DESCRIPTION

The current rule system implementation is very brittle and is unstable. Users are discouraged from using rules at this time.

Create rule is used to define a new rule.

Here, *event* is one of *select*, *update*, *delete* or *insert*. *Object* is either:
  a class name
   or
  class.column

The **from** clause, the **where** clause, and the *action* are respectively normal SQL **from** clauses, **where** clauses and collections of SQL commands with the following change:

> **new** or **current** can appear instead of an instance variable whenever an instance variable is permissible in SQL.

The semantics of a rule is that at the time an individual instance is accessed, updated, inserted or deleted, there is a **current** instance (for retrieves, updates and deletes) and a **new** instance (for updates and appends). If the event specified in the **on** clause and the condition specified in the **where** clause are true for the current instance, then the *action* part of the rule is executed. First, however, values from fields in the current instance and/or the new instance are substituted for:
  current.attribute-name
  new.attribute-name

The *action* part of the rule executes with same command and transaction identifier as the user command that caused activation.

A note of caution about SQL rules is in order. If the same class name or instance variable appears in the event, **where** clause and the *action* parts of a rule, they are all considered different tuple variables. More accurately, **new** and **current** are the only tuple variables that are shared between these clauses. For example, the following two rules have the same semantics:

```
on update to EMP.salary where EMP.name = "Joe"
      do update EMP ( ... ) where ...

on update to EMP-1.salary where EMP-2.name = "Joe"
      do update EMP-3 ( ... ) where ...
```

Each rule can have the optional tag **instead**. Without this tag *action* will be performed in addition to the user command when the event in the condition part of the rule occurs. Alternately, the *action* part will be done instead of the user command. In this later case, the action can be the keyword **nothing**.

When choosing between the rewrite and instance rule systems for a particular rule application, remember that in the rewrite system **current** refers to a relation and some qualifiers whereas in the instance system it refers to an instance (tuple).

It is very important to note that the **rewrite** rule system will neither detect nor process circular rules. For example, though each of the following two rule definitions are accepted by POSTGRES95, the *retrieve* command will cause POSTGRES95 to *crash*:

```
--
--Example of a circular rewrite rule combination.
--
create rule bad_rule_combination_1 is
      on select to EMP
      do instead select to TOYEMP

create rule bad_rule_combination_2 is
      on select to TOYEMP
      do instead select to EMP

--
--This attempt to retrieve from EMP will cause POSTGRES95 to crash.
--
select * from EMP
```

You must have *rule definition* access to a class in order to define a rule on it (see *change acl*(commands).

EXAMPLES

```
--
--Make Sam get the same salary adjustment as Joe
--
create rule example_1 is
    on update EMP.salary where current.name = "Joe"
    do update EMP (salary = new.salary)
      where EMP.name = "Sam"
```

At the time Joe receives a salary adjustment, the event will become true and Joe's current instance and proposed new instance are available to the execution routines. Hence, his new salary is substituted into the *action* part of the rule which is subsequently executed. This propagates Joe's salary on to Sam.

```
--
--Make Bill get Joe's salary when it is accessed
--
create rule example_2 is
    on select to EMP.salary
        where current.name = "Bill"
    do instead
      select (EMP.salary) from EMP where EMP.name = "Joe"


--
--Deny Joe access to the salary of employees in the shoe
--department.  (pg_username() returns the name of the current user)
--
create rule example_3 is
    on select to EMP.salary
      where current.dept = "shoe"
```

```
                  and pg_username() = "Joe"
        do instead nothing


    --
    --Create a view of the employees working in the toy department.
    --
    create TOYEMP(name = char16, salary = int4)

    create rule example_4 is
        on select to TOYEMP
        do instead select (EMP.name, EMP.salary) from EMP
          where EMP.dept = "toy"


    --
    --All new employees must make 5,000 or less
    --
    create rule example_5 is
          on insert to EMP where new.salary > 5000
          do update newset salary = 5000
```

SEE ALSO

drop rule(commands), create view(commands).

BUGS

instead rules do not work properly.

The object in a SQL rule cannot be an array reference and cannot have parameters.

Aside from the "oid" field, system attributes cannot be referenced anywhere in a rule. Among other things, this means that functions of instances (e.g., "foo(emp)" where "emp" is a class) cannot be called anywhere in a rule.

The rule system store the rule text and query plans as text attributes. This implies that creation of rules may fail if the rule plus its various internal representations exceed some value that is on the order of one page (8KB).

NAME

create table — create a new class

SYNOPSIS

create table classname **(**attname-1 type-1 {**,** attname-i type-i}**)**
[**inherits** (classname-1 {**,** classname-i})]
[**archive =** archive_mode]
[**store =** "smgr_name"]
[**arch_store =** "smgr_name"]

DESCRIPTION

Create table will enter a new class into the current data base. The class will be "owned" by the user issuing the command. The name of the class is *classname* and the attributes are as specified in the list of *attname*s. The *i*th attribute is created with the type specified by *type*-i. Each type may be a simple type, a complex type (set) or an array type.

Each array attribute stores arrays that must have the same number of dimensions but may have different sizes and array index bounds. An array of dimension *n* is specified by appending *n* pairs of square brackets:

```
att_name = type[][]..[]
```

The optional **inherits** clause specifies a collection of class names from which this class automatically inherits all fields. If any inherited field name appears more than once, POSTGRES95 reports an error. POSTGRES95 automatically allows the created class to inherit functions on classes above it in the inheritance hierarchy. Inheritance of functions is done according to the conventions of the Common Lisp Object System (CLOS).

Each new class *classname* is automatically created as a type. Therefore, one or more instances from the class are automatically a type and can be used in *alter*table(commands) or other **create table** statements. See *introduction*(commands) for a further discussion of this point.

The optional **store** and **arch_store** keywords may be used to specify a storage manager to use for the new class. The released version of POSTGRES95 supports only "magnetic disk" as a storage manager name; the research system at UC Berkeley provides additional storage managers. **Store** controls the location of current data, and **arch_store** controls the location of historical data. **Arch_store** may only be specified if **archive** is also specified. If either **store** or **arch_store** is not declared, it defaults to "magnetic disk".

The new class is created as a heap with no initial data. A class can have no more than 1600 attributes (realistically, this is limited by the fact that tuple sizes must be less than 8192 bytes), but this limit may be configured lower at some sites. A class cannot have the same name as a system catalog class.

The **archive** keyword specifies whether historical data is to be saved or discarded. *Arch_mode* may be one of:

none No historical access is supported.

light Historical access is allowed and optimized for light update activity.

heavy Historical access is allowed and optimized for heavy update activity.

Arch_mode defaults to "none". Once the archive status is set, there is no way to change it. For details of the optimization, see [STON87].

EXAMPLES

```
--
-- Create class emp with attributes name, sal and bdate
--
```

```
create table emp (name char16, salary float4, bdate abstime)


--
--Create class permemp with pension information that
--inherits all fields of emp
--
create table permemp (plan char16) inherits (emp)


--
--Create class foo on magnetic disk and archive historical data
--
create table foo (bar int4) archive = heavy
    store = "magnetic disk"


--
--Create class tictactoe to store noughts-and-crosses
--boards as a 2-dimensional array
--
create table tictactoe (game int4, board = char[][])


--
--Create a class newemp with a set attribute "manager".  A
--set (complex) attribute may be of the same type as the
--relation being defined (as here) or of a different complex
--type.  The type must exist in the "pg_type" catalog or be
--the one currently being defined.
--
create table newemp (name text, manager = newemp)
```

SEE ALSO

drop table(commands).

NAME

create type — define a new base data type

SYNOPSIS

create type typename **(internallength** = (number **|** **variable**),
[**externallength** = (number **|** **variable**)**,**]
input = input_function,
output = output_function
[**, element** = typename]
[**, delimiter** = <character>]
[**, default** = "string"]
[**, send** = send_function]
[**, receive** = receive_function]
[**, passedbyvalue**]**)**

DESCRIPTION

Create type allows the user to register a new user data type with POSTGRES95 for use in the current data base. The user who defines a type becomes its owner. *Typename* is the name of the new type and must be unique within the types defined for this database.

Create type requires the registration of two functions (using *create function*(commands)) before defining the type. The representation of a new base type is determined by *input_function*, which converts the type's external representation to an internal representation usable by the operators and functions defined for the type. Naturally, *output_function* performs the reverse transformation. Both the input and output functions must be declared to take one or two arguments of type "opaque".

New base data types can be fixed length, in which case **internallength** is a positive integer, or variable length, in which case POSTGRES95 assumes that the new type has the same format as the POSTGRES95-supplied data type, "text". To indicate that a type is variable-length, set **internallength** to *variable*. The external representation is similarly specified using the *externallength* keyword.

To indicate that a type is an array and to indicate that a type has array elements, indicate the type of the array element using the **element** keyword. For example, to define an array of 4 byte integers ("int4"), specify

```
element = int4
```

To indicate the delimiter to be used on arrays of this type, **delimiter** can be set to a specific character. The default delimiter is the comma (",") character.

A **default** value is optionally available in case a user wants some specific bit pattern to mean "data not present."

The optional functions *send_function* and *receive_function* are used when the application program requesting POSTGRES95 services resides on a different machine. In this case, the machine on which POSTGRES95 runs may use a different format for the data type than used on the remote machine. In this case it is appropriate to convert data items to a standard form when **send**ing from the server to the client and converting from the standard format to the machine specific format when the server **receive**s the data from the client. If these functions are not specified, then it is assumed that the internal format of the type is acceptable on all relevant machine architectures. For example, single characters do not have to be converted if passed from a Sun-4 to a DECstation, but many other types do.

The optional **passedbyvalue** flag indicates that operators and functions which use this data type should be passed an argument by value rather than by reference. Note that only types whose internal representation is at most four bytes may be passed by value.

For new base types, a user can define operators, functions and aggregates using the appropriate facilities described in this section.

ARRAY TYPES

Two generalized built-in functions, **array_in** and **array_out,** exist for quick creation of variable-length array types. These functions operate on arrays of any existing POSTGRES95 type.

LARGE OBJECT TYPES

A "regular" POSTGRES95 type can only be 8192 bytes in length. If you need a larger type you must create a Large Object type. The interface for these types is discussed at length in Section 7, the large object interface. The length of all large object types is always *variable,* meaning the **internallength** for large objects is always -1.

EXAMPLES

```
--
--This command creates the box data type and then uses the
--type in a class definition
--
create type box (internallength = 8,
      input = my_procedure_1, output = my_procedure_2)

create table MYBOXES (id = int4, description = box)


--
--This command creates a variable length array type with
--integer elements.
--
create type int4array
   (input = array_in, output = array_out,
    internallength = variable, element = int4)

create table MYARRAYS (id = int4, numbers = int4array)


--
--This command creates a large object type and uses it in
--a class definition.
--
create type bigobj
   (input = lo_filein, output = lo_fileout,
    internallength = variable)

create table BIG_OBJS (id = int4, obj = bigobj)
```

RESTRICTIONS

Type names cannot begin with the underscore character ("_") and can only be 15 characters long. This is because POSTGRES95 silently creates an array type for each base type with a name consisting of the base type's name prepended with an underscore.

SEE ALSO

create function(commands), create operator(commands), drop type(commands), introduction(large objects).

NAME

create version — construct a version class

SYNOPSIS

create version classname1 **from** classname2 [**[**abstime**]**]

DESCRIPTION

Currently, the versioning facility is not working.

This command creates a version class *classname1* which is related to its parent class, *classname2*. Initially, *classname1* has the same contents as *classname2*. As updates to *classname1* occur, however, the content of *classname1* diverges from *classname2*. On the other hand, any updates to *classname2* show transparently through to *classname1*, unless the instance in question has already been updated in *classname1*.

If the optional *abstime* clause is specified, then the version is constructed relative to a **snapshot** of *classname2* as of the time specified.

POSTGRES95 uses the query rewrite rule system to ensure that *classname1* is differentially encoded relative to *classname2*. Moreover, *classname1* is automatically constructed to have the same indexes as *classname2*. It is legal to cascade versions arbitrarily, so a tree of versions can ultimately result. The algorithms that control versions are explained in [ONG90].

EXAMPLE

```
--
--create a version foobar from a snapshot of
--barfoo as of January 17, 1990
--
create version foobar from barfoo [ "Jan 17 1990" ]
```

SEE ALSO

create view(commands), merge(commands).

BUGS

Snapshots (i.e., the optional *abstime* clause) are not implemented in POSTGRES95.

NAME

create view — construct a virtual class

SYNOPSIS

create view view_name **as**
  select expression1 [**as** attr_name1]
  {, expression_i [**as** attr_namei]}
  [**from** from.last]
  [**where** qual]

DESCRIPTION

create view will define a view of a class. This view is not physically materialized; instead the rule system is used to support view processing as in [STON90]. Specifically, a query rewrite retrieve rule is automatically generated to support retrieve operations on views. Then, the user can add as many update rules as desired to specify the processing of update operations to views. See [STON90] for a detailed discussion of this point.

EXAMPLE

```
--
--create a view consisting of toy department employees
--
create view toyemp as
      select e.name
      from emp e
      where e.dept = 'toy'


--
--Specify deletion semantics for toyemp
--
create rule example1 as
      on delete to toyemp
      do instead delete emp
      where emp.oid = current.oid
```

SEE ALSO

create table(commands), create rule(commands),

NAME

create database — create a new database

SYNOPSIS

create database dbname

DESCRIPTION

Create database creates a new POSTGRES95 database. The creator becomes the administrator of the new database.

SEE ALSO

createdb(unix), drop database(commands), destroydb(unix), initdb(unix).

BUGS

This command should **NOT** be executed interactively. The *createdb*(unix) script should be used instead.

NAME

delete — delete instances from a class

SYNOPSIS

delete instance_variable [**from** from_list] [**where** qual]

DESCRIPTION

Delete removes instances which satisfy the qualification, *qual*, from the class specified by *instance_variable*. *Instance_variable* is either a class name or a variable assigned by *from_list*. If the qualification is absent, the effect is to delete all instances in the class. The result is a valid, but empty class.

You must have write access to the class in order to modify it, as well as read access to any class whose values are read in the qualification (see *change acl*(commands).

EXAMPLE

```
--
--Remove all employees who make over $30,000
--
delete from emp where emp.sal > 30000


--
--Clear the hobbies class
--
delete from hobbies
```

SEE ALSO

drop(commands).

NAME

drop database — destroy an existing database

SYNOPSIS

drop database dbname

DESCRIPTION

Drop database removes the catalog entries for an existing database and deletes the directory containing the data. It can only be executed by the database administrator (see *createdb*(commands) for details).

SEE ALSO

create database(commands), destroydb(unix).

BUGS

This query should **NOT** be executed interactively. The *destroydb*(unix) script should be used instead.

NAME

drop table — destroy existing classes

SYNOPSIS

drop table classname-1 { **,** classname-i }

DESCRIPTION

Drop Table removes classes from the data base. Only its owner may destroy a class. A class may be emptied of instances, but not destroyed, by using *delete*(commands).

If a class being destroyed has secondary indices on it, then they will be removed first. The removal of just a secondary index will not affect the indexed class.

The destruction of classes is not reversable. Thus, a destroyed class will not be recovered if a transaction which destroys this class fails to commit. In addition, historical access to instances in a destroyed class is not possible.

EXAMPLE

```
--
--Destroy the emp class
--
drop table emp


--
--Destroy the emp and parts classes
--
drop table emp, parts
```

SEE ALSO

delete(commands), drop index(commands).

NAME

drop aggregate — remove the definition of an aggregate

SYNOPSIS

drop aggregate aggname

DESCRIPTION

drop aggregate will remove all reference to an existing aggregate definition. To execute this command the current user must be the the owner of the aggregate.

EXAMPLE

```
--
--Remove the average aggregate
--
drop aggregate avg
```

SEE ALSO

create aggregate(commands).

NAME

drop function — remove a user-defined C function

SYNOPSIS

drop function function_name (**[type-1 { , type-n }]**)

DESCRIPTION

drop function will remove references to an existing C function. To execute this command the user must be the owner of the function. The input argument types to the function must be specified, as only the function with the given name and argument types will be removed.

EXAMPLE

```
--
--this command removes the square root function
--
drop function sqrt(int4)
```

SEE ALSO

create function(commands).

BUGS

No checks are made to ensure that types, operators or access methods that rely on the function have been removed first.

NAME

drop index — removes an index from POSTGRES95

SYNOPSIS

drop index index_name

DESCRIPTION

This command drops an existing index from the POSTGRES95 system. To execute this command you must be the owner of the index.

EXAMPLE

```
--
--this command will remove the "emp_index" index
--
drop index emp_index
```

SEE ALSO

create index(commands).

NAME

drop operator — remove an operator from the system

SYNOPSIS

drop operator opr_desc

DESCRIPTION

This command drops an existing operator from the database. To execute this command you must be the owner of the operator.

Opr_desc is the name of the operator to be removed followed by a parenthesized list of the operand types for the operator. The left or right type of a left or right unary operator, respectively, may be specified as *none*.

It is the user's responsibility to remove any access methods, operator classes, etc. that rely on the deleted operator.

EXAMPLE

```
--
--Remove power operator a^n for int4
--
drop operator ^ (int4, int4)


--
--Remove left unary operator !a for booleans
--
drop operator ! (none, bool)


--
--Remove right unary factorial operator a! for int4
--
drop operator ! (int4, none)
```

SEE ALSO

create operator(commands).

NAME

drop rule − removes a current rule from POSTGRES95

SYNOPSIS

drop rule rule_name

DESCRIPTION

This command drops the rule named rule_name from the specified POSTGRES95 rule system. POSTGRES95 will immediately cease enforcing it and will purge its definition from the system catalogs.

EXAMPLE

```
--
--This example drops the rewrite rule example_1
--
drop rule example_1
```

SEE ALSO

create rule(commands), drop view(commands).

BUGS

Once a rule is dropped, access to historical information the rule has written may disappear.

NAME

drop type — remove a user-defined type from the system catalogs

SYNOPSIS

drop type typename

DESCRIPTION

This command removes a user type from the system catalogs. Only the owner of a type can remove it.

It is the user's responsibility to remove any operators, functions, aggregates, access methods, subtypes, classes, etc. that use a deleted type.

EXAMPLE

```
--
--remove the box type
--
drop type box
```

SEE ALSO

introduction(commands), create type(commands), drop operator(commands).

BUGS

If a built-in type is removed, the behavior of the backend is unpredictable.

NAME

end — commit the current transaction

SYNOPSIS

end [transaction]

DESCRIPTION

This commands commits the current transaction. All changes made by the transaction become visible to others and are guaranteed to be durable if a crash occurs.

SEE ALSO

abort(commands), begin(commands).

NAME

fetch — fetch instance(s) from a cursor

SYNOPSIS

fetch [(**forward** | **backward**)] [(number | **all**)] [**in** cursor_name]

DESCRIPTION

Fetch allows a user to retrieve instances from a cursor named *cursor_name.* The number of instances retrieved is specified by *number*. If the number of instances remaining in the cursor is less than *number*, then only those available are fetched. Substituting the keyword *all* in place of a number will cause all remaining instances in the cursor to be retrieved. Instances may be fetched in both *forward* and *backward* directions. The default direction is *forward*.

Updating data in a cursor is not supported by POSTGRES95, because mapping cursor updates back to base classes is impossible in general as with view updates. Consequently, users must issue explicit replace commands to update data.

Portals may only be used inside of transaction blocks marked by *begin*(commands) and *end*(commands) because the data that they store spans multiple user queries.

EXAMPLE

```
--
--set up and use a cursor
--
begin
   declare myportal cursor for
   select * from pg-user
end


--
--Fetch all the instances available in the portal FOO
--
fetch all in FOO


--
--Fetch 5 instances backward in the portal FOO
--
fetch backward 5 in FOO
```

SEE ALSO

begin(commands), end(commands), close(commands), move(commands), select(commands).

BUGS

Currently, the smallest transaction in POSTGRES95 is a single SQL command. It should be possible for a single fetch to be a transaction.

NAME

insert — insert tuples to a relation

SYNOPSIS

insert into classname
[(att.expr-1,{att_expr.i})]
{**values** (expression1 {,expression-i}) |
select expression1,{expression-i}
[**from** from-list] [**where** qual]

DESCRIPTION

Insert adds instances that satisfy the qualification, *qual*, to *classname*. *Classname* must be the name of an existing class. The target list specifies the values of the fields to be appended to *classname*. That is, each *att_expr* specifies a field (either an attribute name or an attribute name plus an array specification) to which the corresponding *expression* should be assigned. The fields in the target list may be listed in any order. Fields of the result class which do not appear in the target list default to NULL. If the expression for each field is not of the correct data type, automatic type coercion will be attempted.

An array initialization may take exactly one of the following forms:

```
--
-- Specify a lower and upper index for each dimension
--
att_name[lIndex-1:uIndex-1]..[lIndex-i:uIndex-i] = array_str

--
--Specify only the upper index for each dimension
--(each lower index defaults to 1)
--
att_name[uIndex-1]..[uIndex-i] = array_str

--
--Use the upper index bounds as specified within array_str
--(each lower index defaults to 1)
--
att_name = array_str
```

where each *lIndex* or *uIndex* is an integer constant and *array_str* is an array constant (see *introduction*(commands)).

If the user does not specify any array bounds (as in the third form) then POSTGRES95 will attempt to deduce the actual array bounds from the contents of *array_str*.

If the user does specify explicit array bounds (as in the first and second forms) then the array may be initialized partly or fully using a C-like syntax for array initialization. However, the uninitialized array elements will contain garbage.

You must have write or append access to a class in order to append to it, as well as read access on any class whose values are read in the target list or qualification (see *change acl*(commands)).

EXAMPLES

```
--
--Make a new employee Jones work for Smith
```

```
--
insert into emp
    select newemp.name, newemp.salary,
      "Smith", 1990-newemp.age
      from newemp
      where name = "Jones"


--
--Insert into newemp class to newemp
--
insert into newemp
    select * from newemp1


--
--Create an empty 3x3 gameboard for noughts-and-crosses
--(all of these queries create the same board attribute)
--
insert into tictactoe (game, board[1:3][1:3])
    values(1,'{{"","",""},{},{"",""}}')
insert into tictactoe (game, board[3][3])
    values (2,'{}')
insert into tictactoe (game, board)
    values (3,'{{,,},{,,},{,,}}')
```

SEE ALSO

create table(commands), create type(commands), update(commands), select(commands)

NAME

listen — listen for notification on a relation

SYNOPSIS

listen class_name

DESCRIPTION

listen is used to register the current backend as a listener on the relation *class_name*. When the command **notify** *class_name* is called either from within a rule or at the query level, the frontend applications corresponding to the listening backends are notified. When the backend process exits, this registration is cleared.

This event notification is performed through the LIBPQ protocol and frontend application interface. The application program must explicitly poll a LIBPQ global variable, *PQAsyncNotifyWaiting*, and call the routine *PQnotifies* in order to find out the name of the class to which a given notification corresponds. If this code is not included in the application, the event notification will be queued and never be processed.

SEE ALSO

create rule(commands), notify(commands), select(commands), libpq.

BUGS

There is no way to un-**listen** except to drop the connection (i.e., restart the backend server).

The *monitor*(unix) command does not poll for asynchronous events.

NAME

load — dynamically load an object file

SYNOPSIS

load "filename"

DESCRIPTION

Load loads an object (or ".o") file into POSTGRES95's address space. Once a file is loaded, all functions in that file can be accessed. This function is used in support of ADT's.

If a file is not loaded using the **load** command, the file will be loaded automatically the first time the function is called by POSTGRES95. **Load** can also be used to reload an object file if it has been edited and recompiled. Only objects created from C language files are supported at this time.

EXAMPLE

```
--
--Load the file /usr/postgres/demo/circle.o
--
load "/usr/postgres/demo/circle.o"
```

CAVEATS

Functions in loaded object files should not call functions in other object files loaded through the **load** command, meaning, for example, that all functions in file A should call each other, functions in the standard or math libraries, or in POSTGRES95 itself. They should not call functions defined in a different loaded file B. This is because if B is reloaded, the POSTGRES95 loader is not “smart” enough to relocate the calls from the functions in A into the new address space of B. If B is not reloaded, however, there will not be a problem.

On DECstations, you must use */bin/cc* with the “-G 0” option when compiling object files to be loaded.

Note that if you are porting POSTGRES95 to a new platform, the **load** command will have to work in order to support ADTs.

NAME

notify — signal all frontends and backends listening on a class

SYNOPSIS

notify class_name

DESCRIPTION

notify is used to awaken all backends and consequently all frontends that have executed *listen*(commands) on *class_name*. This can be used either within an instance-level rule as part of the action body or from a normal query. When used from within a normal query, this can be thought of as interprocess communication (IPC). When used from within a rule, this can be thought of as an alerter mechanism.

Notice that the mere fact that a **notify** has been executed does not imply anything in particular about the state of the class (e.g., that it has been updated), nor does the notification protocol transmit any useful information other than the class name. Therefore, all **notify** does is indicate that some backend wishes its peers to examine *class_name* in some application-specific way.

This event notification is performed through the LIBPQ protocol and frontend application interface. The application program must explicitly poll a LIBPQ global variable, *PQAsyncNotifyWaiting*, and call the routine *PQnotifies* in order to find out the name of the class to which a given notification corresponds. If this code is not included in the application, the event notification will be queued and never be processed.

SEE ALSO

define rule(commands), listen(commands), libpq.

NAME

purge — discard historical data

SYNOPSIS

purge classname [**before** abstime] [**after** reltime]

DESCRIPTION

Purge allows a user to specify the historical retention properties of a class. If the date specified is an absolute time such as “Jan 1 1987”, POSTGRES95 will discard tuples whose validity expired before the indicated time. **Purge** with no *before* clause is equivalent to “purge before now”. Until specified with a purge command, instance preservation defaults to “forever”.

The user may purge a class at any time as long as the purge date never decreases. POSTGRES95 will enforce this restriction, silently.

Note that the **purge** command does not do anything except set a parameter for system operation. Use *vacuum*(commands) to enforce this parameter.

EXAMPLE

```
--
--Always discard data in the EMP class
--prior to January 1, 1989
--
purge EMP before "Jan 1 1989"


--
--Retain only the current data in EMP
--
purge EMP
```

SEE ALSO

vacuum(commands).

BUGS AND CAVEATS

Error messages are quite unhelpful. A complaint about “inconsistent times” followed by several nine-digit numbers indicates an attempt to “back up” a purge date on a relation.

You cannot purge certain system catalogs (namely, “pg_class”, “pg_attribute”, “pg_am”, and “pg_amop”) due to circularities in the system catalog code.

This definition of the **purge** command is really only useful for non-archived relations, since tuples will not be discarded from archive relations (they are never vacuumed).

NAME

drop view — removes a view from POSTGRES95

SYNOPSIS

drop view view_name

DESCRIPTION

This command drops an existing view from the POSTGRES95 system. To execute this command you must be the owner of the view.

EXAMPLE

```
/*
 * this command will remove the "myview" view
 */
drop view myview
```

SEE ALSO

create view(commands), drop rule(commands),

NAME

rename — rename a class or an attribute in a class

SYNOPSIS

alter table classname1
  rename to classname2
alter table classname1 [*]
  rename [column] attname1 **to** attname2

DESCRIPTION

The **rename** command causes the name of a class or attribute to change without changing any of the data contained in the affected class. Thus, the class or attribute will remain of the same type and size after this command is executed.

In order to rename an attribute in each class in an entire inheritance hierarchy, use the *classname* of the superclass and append a “*”. (By default, the attribute will not be renamed in any of the subclasses.) This should **always** be done when changing an attribute name in a superclass. If it is not, queries on the inheritance hierarchy such as

```
select * from super* s
```

will not work because the subclasses will be (in effect) missing an attribute found in the superclass.

You must own the class being modified in order to rename it or part of its schema. Renaming any part of the schema of a system catalog is not permitted.

EXAMPLE

```
--
-- change the emp class to personnel
--
alter table emp rename to personnel


--
-- change the sports attribute to hobbies
--
alter table emp rename column sports to hobbies


--
-- make a change to an inherited attribute
--
alter table person * rename column last_name to family_name
```

BUGS

Execution of historical queries using classes and attributes whose names have changed will produce incorrect results in many situations.

Renaming of types, operators, rules, etc., should also be supported.

NAME

rollback — rollback, e.g. aborts, the current transaction

SYNOPSIS

rollback [transaction|work]

DESCRIPTION

This command rolls back the current transaction and causes all the updates made by the transaction to be discarded.

rollback is functionally equivalent to the *abort* command.

SEE ALSO

abort(commands), begin(commands), end(commands).

NAME

select — retrieve instances from a class

SYNOPSIS

select [distinct]
expression1 [**as** attr_name-1]
{, expression-1 [**as** attr_name-i]}
[**into table** classname]
[**from** from-last]
[**group by** attr_name1 {, attr_name-i....}
[**order by** attr_name1
[**using op1**] {, attr_namei [**using opi**] }

DESCRIPTION

Select will get all instances which satisfy the qualification, *qual*, compute the value of each element in the target list, and either (1) return them to an application program through one of two different kinds of portals or (2) store them in a new class.

If into table class name is specified, the result of the query will be stored in a new class with the indicated name.

The **order by** clause allows a user to specify that he wishes the instances sorted according to the corresponding operator. This operator must be a binary one returning a boolean. Multiple sort fields are allowed and are applied from left to right.

The target list specifies the fields to be retrieved. Each *attr_name* specifies the desired attribute or portion of an array attribute. Thus, each *attr_name* takes the form

```
class_name.att_name
```

or, if the user only desires part of an array,

```
--
--Specify a lower and upper index for each dimension
--(i.e., clip a range of array elements)
--
class_name.att_name[lIndex-1:uIndex-1]..[lIndex-i:uIndex-i]

--
--Specify an exact array element
--
class_name.att_name[uIndex-1]..[uIndex-i]
```

where each *lIndex* or *uIndex* is an integer constant.

When you retrieve an attribute which is of a complex type, the behavior of the system depends on whether you used "nested dots" to project out attributes of the complex type or not. See the examples below.

You must have read access to a class to read its values (see *grant/revoke*(commands).

EXAMPLES

```
--
--Find all employees who make more than their manager
--
select e.name
```

```
   from emp e, emp m
   where e.mgr = m.name
   and e.sal >  m.sal


--
--Retrieve all fields for those employees who make
--more than the average salary
--
select avg(sal) as ave
   into table avgsal from emp;


--
--Retrieve all employee names in sorted order
--
select distinct name
   from emp
   order by name using <


--
--Retrieve all employee names that were valid on 1/7/85
--in sorted order
--
selec name
   from emp['January 7 1985'] e
   order by name using <


--
--Construct a new class, raise, containing 1.1
--times all employee's salaries
--
select 1.1 * emp.salary as salary
    into tables raise
    from emp
```

SEE ALSO

insert(commands), close(commands), create table(commands), fetch(commands), update(commands).

BUGS

Retrieve into does not delete duplicates.

If the backend crashes in the course of executing a **retrieve into**, the class file will remain on disk. It can be safely removed by the database DBA, but a subsequent **retrieve into** to the same name will fail with a cryptic error message about "BlockExtend".

NAME

update — replace values of attributes in a class

SYNOPSIS

update classname **set** attname-1 = expression-1,
{attname-i = expression-i}
[**where** qual]

DESCRIPTION

Update changes the values of the attributes specified in *target_list* for all instances which satisfy the qualification, *qual*. Only the attributes to be modified need appear in *target_list*.

Array references use the same syntax found in *select(commands).* That is, either single array elements, a range of array elements or the entire array may be replaced with a single query.

You must have write access to the class in order to modify it, as well as read access to any class whose values are mentioned in the target list or qualification.

EXAMPLES

```
--
--Give all employees who work for Smith a 10% raise
--
update emp
   set sal = 1.1 * sal
   where mgr = 'Smith'
```

SEE ALSO

create table(commands), select(commands).

NAME

vacuum — vacuum a database

SYNOPSIS

vacuum

DESCRIPTION

Vacuum is the POSTGRES95 vacuum cleaner. It opens every class in the database, moves deleted records to the archive for archived relations, cleans out records from aborted transactions, and updates statistics in the system catalogs. The statistics maintained include the number of tuples and number of pages stored in all classes. Running **vacuum** periodically will increase POSTGRES95's speed in processing user queries.

The open database is the one that is vacuumed.

We recommend that production databases be vacuumed nightly, in order to keep statistics relatively current. The **vacuum** query may be executed at any time, however. In particular, after copying a large class into POSTGRES95 or deleting a large number of records, it may be a good idea to issue a **vacuum** query. This will update the system catalogs with the results of all recent changes, and allow the POSTGRES95 query optimizer to make better choices in planning user queries.

SEE ALSO

vacuum(unix).

SECTION 5 — LIBPQ

DESCRIPTION

LIBPQ is the programmer's interface to POSTGRES95. LIBPQ is a set of library routines that allows queries to pass to the POSTGRES95 backend and instances to return through an IPC channel.

This version of the documentation describes the C interface library. Three short programs are included at the end of this section to show how to write programs that use LIBPQ.

There are several examples of LIBPQ applications in the following directories:

```
../src/test/regress
../src/test/examples
../src/bin/psql
```

Frontend programs which use LIBPQ must include the header file and must link with the library.

CONTROL AND INITIALIZATION

The following environment variables can be used to set up default environment values to avoid hard-coding database names into an application program: sets the default server name. sets additional runtime options for the POSTGRES95 backend. sets the default port for communicating with the POSTGRES95 backend. sets the file or tty on which debugging messages from the backend server are displayed. sets the default POSTGRES95 database name. sets the realm to use with POSTGRES95, if it is different from the local realm. If is set, POSTGRES95 applications will attempt authentication with servers for this realm and use separate ticket files to avoid conflicts with local ticket files. This environment variable is only used if authentication is enabled.

The following routines deal with making a connection to a backend from a C program.

Makes a new connection to a backend.

```
PGconn *PQsetdb(char *pghost,
                char *pgport,
                char *pgoptions,
                char *pgtty,
                char *dbName);
```

If any argument is NULL, then the corresponding environment variable is checked. If the environment variable is also not set, then hardwired defaults are used. always returns a valid PGconn pointer. The (see below) command should be called to ensure that a connection was properly made before queries are sent via the connection. LIBPQ programmers should be careful to maintain the PGconn abstraction. Use the accessor functions below to get at the contents of PGconn. Avoid directly referencing the fields of the PGconn structure as they are subject to change in the future. returns the database name of the connection.

```
char *PQdb(PGconn *conn)
```

returns the host name of the connection.

```
char *PQhost(PGconn *conn)
```

returns the pgoptions used in the connection.

```
char *PQoptions(PGconn *conn)
```

returns the pgport of the connection.

```
char *PQport(PGconn *conn)
```

returns the pgtty of the connection.

```
char *PQtty(PGconn *conn)
```

Returns the status of the connection. The status can be CONNECTION_OK or CONNECTION_BAD.

```
ConnStatusType *PQstatus(PGconn *conn)
```

returns the error message associated with the connection

```
char *PQerrorMessage(PGconn* conn);
```

Close the connection to the backend. Also frees memory used by the PGconn structure. The PGconn pointer should not be used after PQfinish has been called.

```
void PQfinish(PGconn *conn)
```

Reset the communication port with the backend. This function will close the IPC socket connection to the backend and attempt to reestablish a new connection to the same backend.

```
void PQreset(PGconn *conn)
```

Enables tracing of messages passed between the frontend and the backend. The messages are echoed to the debug_port file stream.

```
void PQtrace(PGconn *conn,
             FILE* debug_port);
```

Disables tracing of messages passed between the frontend and the backend.

```
void PQuntrace(PGconn *conn);
```

Submit a query to POSTGRES95. Returns a PGresult pointer if the query was successful or a NULL otherwise. If a NULL is returned, can be used to get more information about the error.

```
PGresult *PQexec(PGconn *conn,
                 char *query);
```

The PGresult structure encapsulates the query result returned by the backend. LIBPQ programmers should be careful to maintain the PGresult abstraction. Use the accessor functions described below to retrieve the results of the query. Avoid directly referencing the fields of the PGresult structure as they are subject to change in the future.

Returns the result status of the query. can return one of the following values:

```
PGRES_EMPTY_QUERY,
PGRES_COMMAND_OK,  /* the query was a command */
PGRES_TUPLES_OK,  /* the query successfully returned tuples */
PGRES_COPY_OUT,
PGRES_COPY_IN,
PGRES_BAD_RESPONSE, /* an unexpected response was received */
PGRES_NONFATAL_ERROR,
PGRES_FATAL_ERROR
```

If the result status is PGRES_TUPLES_OK, then the following routines can be used to retrieve the tuples returned by the query. returns the number of tuples (instances) in the query result.

```
int PQntuples(PGresult *res);
```

returns the number of fields (attributes) in the query result.

```
int PQnfields(PGresult *res);
```

returns the field (attribute) name associated with the given field index. Field indices start at 0.

```
char *PQfname(PGresult *res,
             int field_index);
```

returns the field (attribute) index associated with the given field name.

```
int PQfnumber(PGresult *res,
             char* field_name);
```

returns the field type associated with the given field index. The integer returned is an internal coding of the type. Field indices start at 0.

```
Oid PQftype(PGresult *res,
            int field_num);
```

returns the size in bytes of the field associated with the given field index. If the size returned is -1, the field is a variable length field. Field indices start at 0.

```
int2 PQfsize(PGresult *res,
             int field_index);
```

returns the field (attribute) value. For most queries, the value returned by is a null-terminated ASCII string representation of the attribute value. If the query was a result of a cursor, then the value returned by is the binary representation of the type in the internal format of the backend server. It is the programmer's responsibility to cast and convert the data to the correct C type. The value returned by points to storage that is part of the PGresult structure. One must explicitly copy the value into other storage if it is to be used past the lifetime of the PGresult structure itself.

```
char* PQgetvalue(PGresult *res,
```

```
                    int tup_num,
                    int field_num);
```

returns the length of a field (attribute) in bytes. If the field is a the length returned here does include the size field of the varlena, i.e., it is 4 bytes less.

```
int PQgetlength(PGresult *res,
                int tup_num,
                int field_num);
```

Returns the command status associated with the last query command.

```
char *PQcmdStatus(PGresult *res);
```

Returns a string with the object id of the tuple inserted if the last query is an INSERT command. Otherwise, returns an empty string.

```
char* PQoidStatus(PGresult *res);
```

Prints out all the tuples and, optionally, the attribute names to the specified output stream. The programs and both use for output.

```
void PQdisplayTuples(
       PGresult* res,
       FILE* fout,      /* output stream */
       int fillAlign,   /* pad the fields with spaces? */
       char *fieldSep,  /* string to use as the field separator */
       int printHeader, /* display attribute headers */
       int quiet,       /* print the number of rows returned ? */
       );
```

Frees the storage associated with the PGresult. Every query result should be properly freed when it is no longer used. Failure to do this will result in memory leaks in the frontend application. The PQresult* passed in should be a value which is returned from PQexec(). Calling PQclear() on an uninitialized PQresult pointer will very likely result in a core dump.

```
void PQclear(PQresult *res);
```

POSTGRES95 provides a interface to send function calls to the backend. This is a trapdoor into system internals and can be a potential security hole. Most users will not need this feature.

```
PGresult* PQfn(PGconn* conn,
             int fnid,
             int *result_buf,
             int *result_len,
             int result_is_int,
             PQArgBlock *args,
             int nargs);
```

The argument is the object identifier of the function to be executed. is the buffer in which to load the return value. The caller must have allocated sufficient space to store the return value. The result length will be returned in the storage pointed to by If the result is to be an integer value, than should be set to 1; otherwise it should be set to 0. and specify the arguments to the function.

```
typedef struct {
    int len;
    int isint;
    union {
        int *ptr;
      int integer;
    } u;
} PQArgBlock;
```

always returns a valid PGresult*. The resultStatus should be checked before the result is used. The caller is responsible for freeing the PGresult with when it is not longer needed.

POSTGRES95 supports asynchronous notification via the and commands. A backend registers its interest in a particular relation with the LISTEN command. All backends listening on a particular relation will be notified asynchronously when a NOTIFY of that relation name is executed by another backend. No additional information is passed from the notifier to the listener. Thus, typically, any actual data that needs to be communicated is transferred through the relation.

LIBPQ applications are notified whenever a connected backend has received an asynchronous notification. However, the communication from the backend to the frontend is not asynchronous. Notification comes piggy-backed on other query results. Thus, an application must submit queries, even empty ones, in order to receive notice of backend notification. In effect, the LIBPQ application must poll the backend to see if there is any pending notification information. After the execution of a query, a frontend may call to see if any notification data is available from the backend.

returns the notification from a list of unhandled notifications from the backend. Returns NULL if there are no pending notifications from the backend. behaves like the popping of a stack. Once a notification is returned from it is considered handled and will be removed from the list of notifications.

```
PGnotify* PQNotifies(PGconn *conn);
```

The second sample program gives an example of the use of asynchronous notification.

The command in POSTGRES95 has options to read from or write to the network connection used by LIBPQ. Therefore, functions are necessary to access this network connection directly so applications may take full advantage of this capability.

Reads a newline-terminated line of characters (transmitted by the backend server) into a buffer of size Like this routine copies up to characters into It is like however, in that it converts the terminating newline into a null character. returns EOF at EOF, 0 if the entire line has been read, and 1 if the buffer is full but the terminating newline has not yet been read. Notice that the application must check to see if a new line consists of the single character “.”, which indicates that the backend server has finished sending the results of the command. Therefore, if the application ever expects to receive lines that are more than characters long, the application must be sure to check the return value of very carefully. The code in

```
../src/bin/psql/psql.c
```

contains routines that correctly handle the copy protocol.

```
int PQgetline(PGconn *conn,
              char *string,
              int length)
```

Sends a null-terminated to the backend server. The application must explicitly send the single character "." to indicate to the backend that it has finished sending its data.

```
void PQputline(PGconn *conn,
               char *string);
```

Syncs with the backend. This function waits until the backend has finished the copy. It should either be issued when the last string has been sent to the backend using or when the last string has been received from the backend using It must be issued or the backend may get "out of sync" with the frontend. Upon return from this function, the backend is ready to receive the next query. The return value is 0 on successful completion, nonzero otherwise.

```
int PQendcopy(PGconn *conn);
```

As an example:

```
PQexec(conn, "create table foo (a int4, b char16, d float8)");
PQexec(conn, "copy foo from stdin");
PQputline(conn, "3<TAB>hello world<TAB>4.5\n");
PQputline(conn,"4<TAB>goodbye world<TAB>7.11\n");
...
PQputline(conn,".\n");
PQendcopy(conn);
```

Enable tracing of the frontend/backend communication to a debugging file stream.

```
void PQtrace(PGconn *conn
             FILE *debug_port)
```

Disable tracing started by

```
void PQuntrace(PGconn *conn)
```

If the user has generated the appropriate authentication credentials (e.g., obtaining tickets), the frontend/backend authentication process is handled by without any further intervention. The following routines may be called by LIBPQ programs to tailor the behavior of the authentication process.

Returns a pointer to static space containing whatever name the user has authenticated. Use of this routine in place of calls to or by applications is highly recommended, as it is entirely possible that the authenticated user name is the same as value of the environment variable or the user's entry in

```
char *fe_getauthname(char* errorMessage)
```

Specifies that LIBPQ should use authentication service rather than its compiled-in default. This value is typically taken from a command-line switch.

```
void fe_setauthsvc(char *name,
                   char* errorMessage)
```

Any error messages from the authentication attempts are returned in the errorMessage argument.

The query buffer is 8192 bytes long, and queries over that length will be silently truncated.

```
/*
 * testlibpq.c
 *    Test the C version of LIBPQ, the POSTGRES95 frontend library.
 *
 *
 */
#include <stdio.h>
#include "libpq-fe.h"

void
exit_nicely(PGconn* conn)
{
  PQfinish(conn);
  exit(1);
}

main()
{
  char *pghost, *pgport, *pgoptions, *pgtty;
  char* dbName;
  int nFields;
  int i,j;

/*  FILE *debug; */

  PGconn* conn;
  PGresult* res;

  /* begin, by setting the parameters for a backend connection
     if the parameters are null, then the system will try to use
     reasonable defaults by looking up environment variables
     or, failing that, using hardwired constants */
  pghost = NULL;  /* host name of the backend server */
  pgport = NULL;  /* port of the backend server */
  pgoptions = NULL; /* special options to start up the backend server */
  pgtty = NULL;     /* debugging tty for the backend server */
  dbName = "template1";

  /* make a connection to the database */
  conn = PQsetdb(pghost, pgport, pgoptions, pgtty, dbName);

  /* check to see that the backend connection was successfully made */
  if (PQstatus(conn) == CONNECTION_BAD) {
    fprintf(stderr,"Connection to database '%s' failed.0, dbName);
    fprintf(stderr,"%s",PQerrorMessage(conn));
    exit_nicely(conn);
  }

/*  debug = fopen("/tmp/trace.out","w");  */
/*   PQtrace(conn, debug);  */
```

```
/* start a transaction block */
res = PQexec(conn,"BEGIN");
if (PQresultStatus(res) != PGRES_COMMAND_OK) {
  fprintf(stderr,"BEGIN command failed0);
  PQclear(res);
  exit_nicely(conn);
}
/* should PQclear PGresult whenever it is no longer needed to avoid
   memory leaks */
PQclear(res);

/* fetch instances from the pg_database, the system catalog of databases*/
res = PQexec(conn,"DECLARE myportal CURSOR FOR select * from pg_database");
if (PQresultStatus(res) != PGRES_COMMAND_OK) {
  fprintf(stderr,"DECLARE CURSOR command failed0);
  PQclear(res);
  exit_nicely(conn);
}
PQclear(res);

res = PQexec(conn,"FETCH ALL in myportal");
if (PQresultStatus(res) != PGRES_TUPLES_OK) {
  fprintf(stderr,"FETCH ALL command didn't return tuples properly0);
  PQclear(res);
  exit_nicely(conn);
}

/* first, print out the attribute names */
nFields = PQnfields(res);
for (i=0; i < nFields; i++) {
  printf("%-15s",PQfname(res,i));
}
printf("0);

/* next, print out the instances */
for (i=0; i < PQntuples(res); i++) {
  for (j=0  ; j < nFields; j++) {
    printf("%-15s", PQgetvalue(res,i,j));
  }
  printf("0);
}

PQclear(res);

/* close the portal */
res = PQexec(conn, "CLOSE myportal");
PQclear(res);

/* end the transaction */
res = PQexec(conn, "END");
```

```
    PQclear(res);

    /* close the connection to the database and cleanup */
    PQfinish(conn);

  /*   fclose(debug); */
  }
```

```
/*
 * testlibpq2.c
 *    Test of the asynchronous notification interface
 *
   populate a database with the following:

CREATE TABLE TBL1 (i int4);

CREATE TABLE TBL2 (i int4);

CREATE RULE r1 AS ON INSERT TO TBL1 DO [INSERT INTO TBL2 values (new.i); NOTIFY TBL2];

 * Then start up this program
 * After the program has begun, do

INSERT INTO TBL1 values (10);

 *
 *
 */
#include <stdio.h>
#include "libpq-fe.h"

void exit_nicely(PGconn* conn)
{
  PQfinish(conn);
  exit(1);
}

main()
{
  char *pghost, *pgport, *pgoptions, *pgtty;
  char* dbName;
  int nFields;
  int i,j;

  PGconn* conn;
  PGresult* res;
  PGnotify* notify;

  /* begin, by setting the parameters for a backend connection
     if the parameters are null, then the system will try to use
     reasonable defaults by looking up environment variables
     or, failing that, using hardwired constants */
  pghost = NULL;  /* host name of the backend server */
  pgport = NULL;  /* port of the backend server */
  pgoptions = NULL; /* special options to start up the backend server */
  pgtty = NULL;     /* debugging tty for the backend server */
  dbName = getenv("USER"); /* change this to the name of your test database*/
```

```
  /* make a connection to the database */
  conn = PQsetdb(pghost, pgport, pgoptions, pgtty, dbName);

  /* check to see that the backend connection was successfully made */
  if (PQstatus(conn) == CONNECTION_BAD) {
    fprintf(stderr,"Connection to database '%s' failed.0, dbName);
    fprintf(stderr,"%s",PQerrorMessage(conn));
    exit_nicely(conn);
  }

  res = PQexec(conn, "LISTEN TBL2");
  if (PQresultStatus(res) != PGRES_COMMAND_OK) {
    fprintf(stderr,"LISTEN command failed0);
    PQclear(res);
    exit_nicely(conn);
  }
  /* should PQclear PGresult whenever it is no longer needed to avoid
     memory leaks */
  PQclear(res);

  while (1) {
      /* async notification only come back as a result of a query*/
      /* we can send empty queries */
      res = PQexec(conn, " ");
/*      printf("res->status = %s0, pgresStatus[PQresultStatus(res)]); */
      /* check for asynchronous returns */
      notify = PQnotifies(conn);
      if (notify) {
        fprintf(stderr,
              "ASYNC NOTIFY of '%s' from backend pid '%d' received0,
              notify->relname, notify->be_pid);
        free(notify);
        break;
      }
      PQclear(res);
  }

  /* close the connection to the database and cleanup */
  PQfinish(conn);

}
```

```
/*
 * testlibpq3.c
 *    Test the C version of LIBPQ, the POSTGRES95 frontend library.
 *   tests the binary cursor interface
 *
 *
 *
 populate a database by doing the following:

CREATE TABLE test1 (i int4, d float4, p polygon);

INSERT INTO test1 values (1, 3.567, '(3.0, 4.0, 1.0, 2.0)'::polygon);

INSERT INTO test1 values (2, 89.05, '(4.0, 3.0, 2.0, 1.0)'::polygon);

 the expected output is:

tuple 0: got
 i = (4 bytes) 1,
 d = (4 bytes) 3.567000,
 p = (4 bytes) 2 points         boundbox = (hi=3.000000/4.000000, lo = 1.000000,2.000000)
tuple 1: got
 i = (4 bytes) 2,
 d = (4 bytes) 89.050003,
 p = (4 bytes) 2 points         boundbox = (hi=4.000000/3.000000, lo = 2.000000,1.000000)

 *
 */
#include <stdio.h>
#include "libpq-fe.h"
#include "utils/geo-decls.h" /* for the POLYGON type */

void exit_nicely(PGconn* conn)
{
  PQfinish(conn);
  exit(1);
}

main()
{
  char *pghost, *pgport, *pgoptions, *pgtty;
  char* dbName;
  int nFields;
  int i,j;
  int i_fnum, d_fnum, p_fnum;

  PGconn* conn;
  PGresult* res;

  /* begin, by setting the parameters for a backend connection
```

```
     if the parameters are null, then the system will try to use
     reasonable defaults by looking up environment variables
     or, failing that, using hardwired constants */
  pghost = NULL;  /* host name of the backend server */
  pgport = NULL;  /* port of the backend server */
  pgoptions = NULL; /* special options to start up the backend server */
  pgtty = NULL;     /* debugging tty for the backend server */

  dbName = getenv("USER");  /* change this to the name of your test database*/

  /* make a connection to the database */
  conn = PQsetdb(pghost, pgport, pgoptions, pgtty, dbName);

  /* check to see that the backend connection was successfully made */
  if (PQstatus(conn) == CONNECTION_BAD) {
    fprintf(stderr,"Connection to database '%s' failed.0, dbName);
    fprintf(stderr,"%s",PQerrorMessage(conn));
    exit_nicely(conn);
  }

  /* start a transaction block */
  res = PQexec(conn,"BEGIN");
  if (PQresultStatus(res) != PGRES_COMMAND_OK) {
    fprintf(stderr,"BEGIN command failed0);
    PQclear(res);
    exit_nicely(conn);
  }
  /* should PQclear PGresult whenever it is no longer needed to avoid
     memory leaks */
  PQclear(res);

  /* fetch instances from the pg_database, the system catalog of databases*/
  res = PQexec(conn,"DECLARE mycursor BINARY CURSOR FOR select * from test1");
  if (PQresultStatus(res) != PGRES_COMMAND_OK) {
    fprintf(stderr,"DECLARE CURSOR command failed0);
    PQclear(res);
    exit_nicely(conn);
  }
  PQclear(res);

  res = PQexec(conn,"FETCH ALL in mycursor");
  if (PQresultStatus(res) != PGRES_TUPLES_OK) {
    fprintf(stderr,"FETCH ALL command didn't return tuples properly0);
    PQclear(res);
    exit_nicely(conn);
  }

  i_fnum = PQfnumber(res,"i");
  d_fnum = PQfnumber(res,"d");
  p_fnum = PQfnumber(res,"p");
```

```
    for (i=0;i<3;i++) {
        printf("type[%d] = %d, size[%d] = %d0,
             i, PQftype(res,i),
             i, PQfsize(res,i));
    }
    for (i=0; i < PQntuples(res); i++) {
      int *ival;
      float *dval;
      int plen;
      POLYGON* pval;
      /* we hard-wire this to the 3 fields we know about */
      ival =  (int*)PQgetvalue(res,i,i_fnum);
      dval =  (float*)PQgetvalue(res,i,d_fnum);
      plen = PQgetlength(res,i,p_fnum);

      /* plen doesn't include the length field so need to increment by VARHDSZ*/
      pval = (POLYGON*) malloc(plen + VARHDRSZ);
      pval->size = plen;
      memmove((char*)&pval->npts, PQgetvalue(res,i,p_fnum), plen);
      printf("tuple %d: got0, i);
      printf(" i = (%d bytes) %d,0,
           PQgetlength(res,i,i_fnum), *ival);
      printf(" d = (%d bytes) %f,0,
           PQgetlength(res,i,d_fnum), *dval);
      printf(" p = (%d bytes) %d points boundbox = (hi=%f/%f, lo = %f,%f)0,
           PQgetlength(res,i,d_fnum),
           pval->npts,
           pval->boundbox.xh,
           pval->boundbox.yh,
           pval->boundbox.xl,
           pval->boundbox.yl);
    }

    PQclear(res);

    /* close the portal */
    res = PQexec(conn, "CLOSE mycursor");
    PQclear(res);

    /* end the transaction */
    res = PQexec(conn, "END");
    PQclear(res);

    /* close the connection to the database and cleanup */
    PQfinish(conn);

  }
```

993/08/23 09:03:16 aoki Exp $

SECTION 6 — LARGE OBJECTS

DESCRIPTION

In POSTGRES95, data values are stored in tuples and individual tuples cannot span data pages. Since the size of a data page is 8192 bytes, the upper limit on the size of a data value is relatively low. To support the storage of larger atomic values, POSTGRES95 provides a large object interface. This interface provides file-oriented access to user data that has been declared to be a large type.

This section describes the implementation and the programmatic and query language interfaces to POSTGRES95 large object data.

Originally, postgres 4.2 supports three standard implementations of large objects: as files external to POSTGRES95, as UNIX files managed by POSTGRES95, and as data stored within the POSTGRES95 database. It causes considerable confusion among users. As a result, we only support large objects as data stored within the POSTGRES95 database in POSTGRES95. Even though is is slower to access, it provides stricter data integrity and time travel. For historical reasons, they are called Inversion large objects. (We will use Inversion and large objects interchangeably to mean the same thing in this section.)

The Inversion large object implementation breaks large objects up into "chunks" and stores the chunks in tuples in the database. A B-tree index guarantees fast searches for the correct chunk number when doing random access reads and writes.

The facilities POSTGRES95 provides to access large objects, both in the backend as part of user-defined functions or the front end as part of an application using the interface, are described below. (For users familiar with postgres 4.2, POSTGRES95 has a new set of functions providing a more coherent interface. The interface is the same for dynamically-loaded C functions as well as for .

The POSTGRES95 large object interface is modeled after the UNIX file system interface, with analogues of etc. User functions call these routines to retrieve only the data of interest from a large object. For example, if a large object type called existed that stored photographs of faces, then a function called could be declared on data. could look at the lower third of a photograph, and determine the color of the beard that appeared there, if any. The entire large object value need not be buffered, or even examined, by the function.

Large objects may be accessed from dynamically-loaded C functions or database client programs that link the LIBPQ library. POSTGRES95 provides a set of routines that support opening, reading, writing, closing, and seeking on large objects.

The routine

```
Oid lo_creat(PGconn *conn, int mode)
```

creates a new large object. The is a bitmask describing several different attributes of the new object. The symbolic constants listed here are defined in

```
/usr/local/postgres95/src/backend/libpq/libpq-fs.h
```

The access type (read, write, or both) is controlled by ing together the bits and If the large object should be archived — that is, if historical versions of it should be moved periodically to a special archive relation — then the bit should be set. The low-order sixteen bits of are the storage manager number on which the large object should reside. For sites other than Berkeley, these bits should always be zero.

The commands below create an (Inversion) large object:

```
inv_oid = lo_creat(INV_READ|INV_WRITE|INV_ARCHIVE);
```

To import a UNIX file as a large object, call

```
Oid
lo_import(PGconn *conn, text *filename)
```

The argument specifies the UNIX pathname of the file to be imported as a large object. To export a large object into UNIX file, call

```
int
lo_export(PGconn *conn, Oid lobjId, text *filename)
```

The argument specifies the Oid of the large object to export and the argument specifies the UNIX pathname of the file.

To open an existing large object, call

```
int
lo_open(PGconn *conn, Oid lobjId, int mode, ...)
```

The argument specifies the Oid of the large object to open. The mode bits control whether the object is opened for reading writing INV_WRITE), (or both.

A large object cannot be opened before it is created. returns a large object descriptor for later use in and

The routine

```
int
lo_write(PGconn *conn, int fd, char *buf, int len)
```

writes bytes from to large object The argument must have been returned by a previous

The number of bytes actually written is returned. In the event of an error, the return value is negative.

To change the current read or write location on a large object, call

```
int
lo_lseek(PGconn *conn, int fd, int offset, int whence)
```

This routine moves the current location pointer for the large object described by to the new location specified by The valid values for .i whence are and

A large object may be closed by calling

```
int
lo_close(PGconn *conn, int fd)
```

where is a large object descriptor returned by On success, returns zero. On error, the return value is negative.

There are two built-in registered functions, and which are convenient for use in SQL queries.

Here is an example of there use

```
CREATE TABLE image (
        name            text,
        raster          oid
);

INSERT INTO image (name, raster)
   VALUES ('beautiful image', lo_import('/etc/motd'));

SELECT lo_export(image.raster, "/tmp/motd") from image
   WHERE name = 'beautiful image';
```

Below is a sample program which shows how the large object interface in

can be used. Parts of the program are commented out but are left in the source for the readers benefit. This program can be found in

```
../src/test/examples
```

Frontend applications which use the large object interface in .br

should include the header file and link with the library.

```
/*-------------------------------------------------------------------------
 *
 * testlo.c--
 *    test using large objects with libpq
 *
 * Copyright (c) 1994, Regents of the University of California
 *
 *
 * IDENTIFICATION
 *    $Header: /usr/local/devel/pglite/cvs/src/doc/ref/large_objects.3pqsrc,v 1.1 1996/02/24 01:16:24 jolly Exp $
 *
 *-------------------------------------------------------------------------
 */
#include <stdio.h>
#include "libpq-fe.h"
#include "libpq/libpq-fs.h"

#define BUFSIZE          1024

/*
 * importFile -
 *    import file "in_filename" into database as large object "lobjOid"
 *
 */
Oid importFile(PGconn *conn, char *filename)
{
    Oid lobjId;
    int lobj_fd;
    char buf[BUFSIZE];
    int nbytes, tmp;
    int fd;

    /*
     * open the file to be read in
     */
    fd = open(filename, O_RDONLY, 0666);
    if (fd < 0)  {   /* error */
      fprintf(stderr, "can't open unix file
    }

    /*
     * create the large object
     */
    lobjId = lo_creat(conn, INV_READ|INV_WRITE);
    if (lobjId == 0) {
      fprintf(stderr, "can't create large object");
    }

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);
    /*
```

```
      * read in from the Unix file and write to the inversion file
      */
     while ((nbytes = read(fd, buf, BUFSIZE)) > 0) {
       tmp = lo_write(conn, lobj_fd, buf, nbytes);
       if (tmp < nbytes) {
           fprintf(stderr, "error while reading
       }
     }

     (void) close(fd);
     (void) lo_close(conn, lobj_fd);

     return lobjId;
 }

 void pickout(PGconn *conn, Oid lobjId, int start, int len)
 {
     int lobj_fd;
     char* buf;
     int nbytes;
     int nread;

     lobj_fd = lo_open(conn, lobjId, INV_READ);
     if (lobj_fd < 0) {
       fprintf(stderr,"can't open large object %d",
             lobjId);
     }

     lo_lseek(conn, lobj_fd, start, SEEK_SET);
     buf = malloc(len+1);

     nread = 0;
     while (len - nread > 0) {
       nbytes = lo_read(conn, lobj_fd, buf, len - nread);
       buf[nbytes] = ' ';
       fprintf(stderr,">>> %s", buf);
       nread += nbytes;
     }
     fprintf(stderr,"0);
     lo_close(conn, lobj_fd);
 }

 void overwrite(PGconn *conn, Oid lobjId, int start, int len)
 {
     int lobj_fd;
     char* buf;
     int nbytes;
     int nwritten;
     int i;
```

```
    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0) {
      fprintf(stderr,"can’t open large object %d",
            lobjId);
    }

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len+1);

    for (i=0;i<len;i++)
      buf[i] = ’X’;
    buf[i] = ’ ’;

    nwritten = 0;
    while (len - nwritten > 0) {
      nbytes = lo_write(conn, lobj_fd, buf + nwritten, len - nwritten);
      nwritten += nbytes;
    }
    fprintf(stderr,"0);
    lo_close(conn, lobj_fd);
}


/*
 * exportFile -
 *    export large object "lobjOid" to file "out_filename"
 *
 */
void exportFile(PGconn *conn, Oid lobjId, char *filename)
{
    int lobj_fd;
    char buf[BUFSIZE];
    int nbytes, tmp;
    int fd;

    /*
     * create an inversion "object"
     */
    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0) {
      fprintf(stderr,"can’t open large object %d",
            lobjId);
    }

    /*
     * open the file to be written to
     */
    fd = open(filename, O_CREAT|O_WRONLY, 0666);
    if (fd < 0)  {   /* error */
      fprintf(stderr, "can’t open unix file
```

```
                filename);
        }

        /*
         * read in from the Unix file and write to the inversion file
         */
        while ((nbytes = lo_read(conn, lobj_fd, buf, BUFSIZE)) > 0) {
          tmp = write(fd, buf, nbytes);
            if (tmp < nbytes) {
              fprintf(stderr,"error while writing
                    filename);
          }
        }

        (void) lo_close(conn, lobj_fd);
        (void) close(fd);

        return;
    }

    void
    exit_nicely(PGconn* conn)
    {
      PQfinish(conn);
      exit(1);
    }

    int
    main(int argc, char **argv)
    {
        char *in_filename, *out_filename;
        char *database;
        Oid lobjOid;
        PGconn *conn;
        PGresult *res;

        if (argc != 4) {
          fprintf(stderr, "Usage: %s database_name in_filename out_filename0,
                argv[0]);
          exit(1);
        }

        database = argv[1];
        in_filename = argv[2];
        out_filename = argv[3];

        /*
         * set up the connection
         */
        conn = PQsetdb(NULL, NULL, NULL, NULL, database);
```

```
    /* check to see that the backend connection was successfully made */
    if (PQstatus(conn) == CONNECTION_BAD) {
      fprintf(stderr,"Connection to database '%s' failed.0, database);
      fprintf(stderr,"%s",PQerrorMessage(conn));
      exit_nicely(conn);
    }

    res = PQexec(conn, "begin");
    PQclear(res);
    printf("importing file
/*  lobjOid = importFile(conn, in_filename); */
    lobjOid = lo_import(conn, in_filename);
/*
    printf("as large object %d.0, lobjOid);

    printf("picking out bytes 1000-2000 of the large object0);
    pickout(conn, lobjOid, 1000, 1000);

    printf("overwriting bytes 1000-2000 of the large object with X's0);
    overwrite(conn, lobjOid, 1000, 1000);
*/

    printf("exporting large object to file
/*    exportFile(conn, lobjOid, out_filename); */
    lo_export(conn, lobjOid,out_filename);

    res = PQexec(conn, "end");
    PQclear(res);
    PQfinish(conn);
    exit(0);
}
```

SECTION 7 — SYSTEM CATALOGS

DESCRIPTION

In this section we list each of the attributes of the system catalogs and define their meanings.

CLASS/TYPE SYSTEM CATALOGS

These catalogs form the core of the extensibility system:

name	**shared/local**	**description**
pg_aggregate	local	aggregate functions
pg_am	local	access methods
pg_amop	local	operators usable with specific access methods
pg_amproc	local	procedures used with specific access methods
pg_attribute	local	class attributes
pg_class	local	classes
pg_index	local	secondary indices
pg_inherits	local	class inheritance hierarchy
pg_language	local	procedure implementation languages
pg_opclass	local	operator classes
pg_operator	local	query language operators
pg_proc	local	procedures (functions)
pg_type	local	data types

ENTITIES

These catalogs deal with identification of entities known throughout the site:

name	**shared/local**	**description**
pg_database	shared	current databases
pg_group	shared	user groups
pg_user	shared	valid users

RULE SYSTEM CATALOGS

name	**shared/local**	**description**
pg_listener	local	processes waiting on alerters
pg_prs2plans	local	instance system procedures
pg_prs2rule	local	instance system rules
pg_prs2stub	local	instance system ‘‘stubs’’
pg_rewrite	local	rewrite system information

LARGE OBJECT CATALOGS

These catalogs are specific to the Inversion file system and large objects in general:

name	**shared/local**	**description**
pg_lobj	local	description of a large object
pg_naming	local	Inversion name space mapping
pg_platter	local	jukebox platter inventory
pg_plmap	local	jukebox platter extent map

INTERNAL CATALOGS

These catalogs are internal classes that are not stored as normal heaps and cannot be accessed through normal means (attempting to do so causes an error).

name	**shared/local**	**description**

pg_log	shared	transaction commit/abort log
pg_magic	shared	magic constant
pg_time	shared	commit/abort times
pg_variable	shared	special variable values

There are several other classes defined with "pg_" names. Aside from those that end in "ind" (secondary indices), these are all obsolete or otherwise deprecated.

CLASS/TYPE SYSTEM CATALOGS

The following catalogs relate to the class/type system.

```
/*
 * aggregates
 *
 * see DEFINE AGGREGATE for an explanation of transition functions
 */
pg_aggregate
    NameData      aggname            /* aggregate name (e.g., "count") */
    oid         aggowner             /* usesysid of creator */
    regproc     aggtransfn1          /* first transition function */
    regproc     aggtransfn2          /* second transition function */
    regproc     aggfinalfn           /* final function */
    oid         aggbasetype          /* type of data on which aggregate
                                        operates */
    oid         aggtranstype1        /* type returned by aggtransfn1 */
    oid         aggtranstype2        /* type returned by aggtransfn2 */
    oid         aggfinaltype         /* type returned by aggfinalfn */
    text        agginitval1          /* external format of initial
                                        (starting) value of aggtransfn1 */
    text        agginitval2          /* external format of initial
                                        (starting) value of aggtransfn2 */


pg_am
    NameData      amname             /* access method name */
    oid         amowner              /* usesysid of creator */
    char        amkind               /* - deprecated */
                                     /* originally:
                                        h=hashed
                                        o=ordered
                                        s=special */
    int2        amstrategies         /* total NUMBER of strategies by which
                                        we can traverse/search this AM */
    int2        amsupport            /* total NUMBER of support functions
                                        that this AM uses */
    regproc     amgettuple           /* "next valid tuple" function */
    regproc     aminsert             /* "insert this tuple" function */
    regproc     amdelete             /* "delete this tuple" function */
    regproc     amgetattr            /* - deprecated */
    regproc     amsetlock            /* - deprecated */
    regproc     amsettid             /* - deprecated */
    regproc     amfreetuple          /* - deprecated */
```

```
    regproc    ambeginscan       /* "start new scan" function */
    regproc    amrescan          /* "restart this scan" function */
    regproc    amendscan         /* "end this scan" function */
    regproc    ammarkpos         /* "mark current scan position"
                                    function */
    regproc    amrestrpos        /* "restore marked scan position"
                                    function */
    regproc    amopen            /* - deprecated */
    regproc    amclose           /* - deprecated */
    regproc    ambuild           /* "build new index" function */
    regproc    amcreate          /* - deprecated */
    regproc    amdestroy         /* - deprecated */


pg_amop
    oid        amopid            /* access method with which this
                                    operator be used */
    oid        amopclaid         /* operator class with which this
                                    operator can be used */
    oid        amopopr           /* the operator */
    int2       amopstrategy      /* traversal/search strategy number
                                    to which this operator applies */
    regproc    amopselect        /* function to calculate the operator
                                    selectivity */
    regproc    amopnpages        /* function to calculate the number of
                                    pages that will be examined */


pg_amproc
    oid        amid              /* access method with which this
                                    procedure is associated */
    oid        amopclaid         /* operator class with which this
                                    operator can be used */
    oid        amproc            /* the procedure */
    int2       amprocnum         /* support function number to which
                                    this operator applies */


pg_class
     NameData    relname         /* class name */
     oid       relowner          /* usesysid of owner */
     oid       relam /* access method */
     int4      relpages          /* # of 8KB pages */
     int4      reltuples         /* # of instances */
     abstime   relexpires        /* time after which instances are
                                    deleted from non-archival storage */
     reltime   relpreserved      /* timespan after which instances are
                                    deleted from non-archival storage */
     bool      relhasindex       /* does the class have a secondary
                                    index? */
```

```
     bool       relisshared          /* is the class shared or local? */
     char       relkind              /* type of relation:
                                        i=index
                                        r=relation (heap)
                                        s=special
                                        u=uncatalogued (temporary) */
     char       relarch              /* archive mode:
                                        h=heavy
                                        l=light
                                        n=none */
     int2       relnatts             /* current # of non-system
                                        attributes */
     int2       relsmgr              /* storage manager:
                                        0=magnetic disk
                                        1=sony WORM jukebox
                                        2=main memory */
     int28      relkey               /* - unused */
     oid8       relkeyop             /* - unused */
     aclitem    relacl[1]            /* access control lists */


pg_attribute
    oid         attrelid             /* class containing this attribute */
    NameData      attname            /* attribute name */
    oid         atttypid             /* attribute type */
    oid         attdefrel            /* - deprecated */
    int4        attnvals             /* - deprecated */
    oid         atttyparg            /* - deprecated */
    int2        attlen               /* attribute length, in bytes
                                        -1=variable */
    int2        attnum               /* attribute number
                                        >0=user attribute
                                        <0=system attribute */
    int2        attbound             /* - deprecated */
    bool        attbyval             /* type passed by value? */
    bool        attcanindex          /* - deprecated */
    oid         attproc              /* - deprecated */
    int4        attnelems            /* # of array dimensions */
    int4        attcacheoff          /* cached offset into tuple */
    bool        attisset             /* is attribute set-valued? */


pg_inherits
    oid         inhrel               /* child class */
    oid         inhparent            /* parent class */
    int4        inhseqno             /* - deprecated */


    oid         indexrelid           /* oid of secondary index class */
    oid         indrelid             /* oid of indexed heap class */
```

```
    oid         indproc              /* function to compute index key from
                                        attribute(s) in heap
                                        0=not a functional index */
    int28       indkey               /* attribute numbers of key
                                        attribute(s) */
    oid8        indclass             /* opclass of each key */
    bool        indisclustered       /* is the index clustered?
                                        - unused */
    bool        indisarchived        /* is the index archival?
                                        - unused */
    text        indpred              /* query plan for partial index
                                        predicate */


pg_type
    NameData      typname            /* type name */
    oid         typowner             /* usesysid of owner */
    int2        typlen               /* length in internal form
                                        -1=variable-length */
    int2        typprtlen            /* length in external form */
    bool        typbyval             /* type passed by value? */
    char        typtype              /* kind of type:
                                        c=catalog (composite)
                                        b=base */
    bool        typisdefined         /* defined or still a shell? */
    char        typdelim             /* delimiter for array external form */
    oid         typrelid             /* class (if composite) */
    oid         typelem              /* type of each array element */
    regproc     typinput             /* external-internal conversion
                                        function */
    regproc     typoutput            /* internal-external conversion
                                        function */
    regproc     typreceive           /* client-server conversion function */
    regproc     typsend              /* server-client conversion function */
    text        typdefault           /* default value */


pg_operator
    NameData      oprname            /* operator name */
    oid         oprowner             /* usesysid of owner */
    int2        oprprec              /* - deprecated */
    char        oprkind              /* kind of operator:
                                        b=binary
                                        l=left unary
                                        r=right unary */
    bool        oprisleft            /* is operator left/right associative? */
    bool        oprcanhash           /* is operator usable for hashjoin? */
    oid         oprleft              /* left operand type */
    oid         oprright             /* right operand type */
    oid         oprresult            /* result type */
```

```
    oid         oprcom              /* commutator operator */
    oid         oprnegate           /* negator operator */
    oid         oprlsortop          /* sort operator for left operand */
    oid         oprrsortop          /* sort operator for right operand */
    regproc     oprcode             /* function implementing this operator */
    regproc     oprrest             /* function to calculate operator
                                       restriction selectivity */
    regproc     oprjoin             /* function to calculate operator
                                       join selectivity */


pg_opclass
    NameData      opcname           /* operator class name */


pg_proc
    NameData      proname           /* function name */
    oid         proowner            /* usesysid of owner */
    oid         prolang             /* function implementation language */
    bool        proisinh            /* - deprecated */
    bool        proistrusted        /* run in server or untrusted function
                                       process? */
    bool        proiscachable       /* can the function return values be
                                       cached? */
    int2        pronargs            /* # of arguments */
    bool        proretset           /* does the function return a set?
                                       - unused */
    oid         prorettype          /* return type */
    oid8        proargtypes         /* argument types */
    int4        probyte_pct         /* % of argument size (in bytes) that
                                       needs to be examined in order to
                                       compute the function */
    int4        properbyte_cpu      /* sensitivity of the function's
                                       running time to the size of its
                                       inputs */
    int4        propercall_cpu      /* overhead of the function's
                                       invocation (regardless of input
                                       size) */
    int4        prooutin_ratio      /* size of the function's output as a
                                       percentage of the size of the input */
    text        prosrc              /* function definition (postquel only) */
    bytea       probin              /* path to object file (C only) */


pg_language
    NameData      lanname           /* language name */
    text        lancompiler         /* - deprecated */
```

ENTITIES

```
pg_database
    NameData     datname          /* database name */
    oid          datdba           /* usesysid of database administrator */
    text         datpath          /* directory of database under
                                     $PGDATA */

pg_group
    NameData     groname          /* group name */
    int2         grosysid         /* group's UNIX group id */
    int2         grolist[1]       /* list of usesysids of group members */

pg_user
    NameData     usename          /* user's name */
    int2         usesysid         /* user's UNIX user id */
    bool         usecreatedb      /* can user create databases? */
    bool         usetrace         /* can user set trace flags? */
    bool         usesuper         /* can user be POSTGRES superuser? */
    bool         usecatupd        /* can user update catalogs? */
```

RULE SYSTEM CATALOGS

```
pg_listener
    NameData     relname          /* class for which asynchronous
                                     notification is desired */
    int4         listenerpid      /* process id of server corresponding
                                     to a frontend program waiting for
                                     asynchronous notification */
    int4         notification     /* whether an event notification for
                                     this process id still pending */

pg_prs2rule
    NameData     prs2name         /* rule name */
    char         prs2eventtype    /* rule event type:
                                     R=retrieve
                                     U=update (replace)
                                     A=append
                                     D=delete */
    oid          prs2eventrel     /* class to which event applies */
    int2         prs2eventattr    /* attribute to which event applies */
    float8       necessary        /* - deprecated */
    float8       sufficient       /* - deprecated */
    text         prs2text         /* text of original rule definition */

pg_prs2plans
    oid          prs2ruleid       /* prs2rule instance for which this
```

```
                                            plan is used */
    int2        prs2planno              /* plan number (one rule may invoke
                                           multiple plans) */
    text        prs2code                /* external representation of the plan */


pg_prs2stub
    oid         prs2relid               /* class to which this rule applies */
    bool        prs2islast              /* is this the last stub fragment? */
    int4        prs2no                  /* stub fragment number */
    stub        prs2stub                /* stub fragment */


pg_rewrite
    NameData      rulename              /* rule name */
    char        ev_type                 /* event type:
                                           RETRIEVE, REPLACE, APPEND, DELETE
                                           codes are parser-dependent (!?) */
    oid         ev_class                /* class to which this rule applies */
    int2        ev_attr                 /* attribute to which this rule applies */
    bool        is_instead              /* is this an "instead" rule? */
    text        ev_qual                 /* qualification with which to modify
                                           (rewrite) the plan that triggered this
                                           rule */
    text        action                  /* parse tree of action */
```

LARGE OBJECT CATALOGS

```
pg_lobj
    oid         ourid                   /* 'ourid' from pg_naming that
                                           identifies this object in the
                                           Inversion file system namespace */
    int4        objtype                 /* storage type code:
                                           0=Inversion
                                           1=Unix
                                           2=External
                                           3=Jaquith */
    bytea       object_descripto/* opaque object-handle structure */


pg_naming
    NameData      filename              /* filename component */
    oid         ourid                   /* random oid used to identify this
                                           instance in other instances (can't
                                           use the actual oid for obscure
                                           reasons */
    oid         parentid                /* pg_naming instance of parent
                                           Inversion file system directory */


pg_platter
```

```
     NameData     plname             /* platter name */
     int4        plstart             /* the highest OCCUPIED extent */


pg_plmap
     oid         plid                /* platter (in pg_platter) on which
                                        this extent (of blocks) resides */
     oid         pldbid              /* database of the class to which this
                                        extent (of blocks) belongs */
     oid         plrelid             /* class to which this extend (of
                                        blocks) belongs */
     int4        plblkno             /* starting block number within the
                                        class */
     int4        ploffset            /* offset within the platter at which
                                        this extent begins */
     int4        plextentsz          /* length of this extent */
```

SECTION 8 — FILES

OVERVIEW

This section describes some of the important files used by POSTGRES95.

NOTATION

“.../” at the front of file names represents the path to the postgres user’s home directory. Anything in square brackets (“**[**” and “**]**”) is optional. Anything in braces (“**{**” and “**}**”) can be repeated 0 or more times. Parentheses (**“**(**”** and **“**)**”**) are used to group boolean expressions. **|** is the boolean operator OR.

BUGS

The descriptions of

```
.../data/PG_VERSION,
.../data/base/*/PG_VERSION,
```

the temporary sort files, and the database debugging trace files are absent.

NAME

.../src/backend/obj/{local,dbdb}.bki — template scripts

DESCRIPTION

Backend Interface (BKI) files are scripts that describe the contents of the initial POSTGRES95 database. This database is constructed during system installation, by the *initdb* command. *Initdb* executes the POSTGRES95 backend with a special set of flags, that cause it to consume the BKI scripts and bootstrap a database.

These files are automatically generated from system header files during installation. They are not intended for use by humans, and you do not need to understand their contents in order to use POSTGRES95. These files are copied to

```
.../files/{global1,local1_XXX}.bki
```

during system installation.

All new user databases will be created by copying the template database that POSTGRES95 constructs from the BKI files. Thus, a simple way to customize the template database is to let the POSTGRES95 initialization script create it for you, and then to run the terminal monitor to make the changes you want.

The POSTGRES95 backend interprets BKI files as described below. This description will be easier to understand if the example in ".../files/global1.bki" is at hand.

Commands are composed of a command name followed by space separated arguments. Arguments to a command which begin with a "$" are treated specially. If "$$" are the first two characters, then the first "$" is ignored and the argument is then processed normally. If the "$" is followed by space, then it is treated as a NULL value. Otherwise, the characters following the "$" are interpreted as the name of a macro causing the argument to be replaced with the macro's value. It is an error for this macro to be undefined.

Macros are defined using

```
define macro macro_name = macro_value
```

and are undefined using

```
undefine macro macro_name
```

and redefined using the same syntax as define.

Lists of general commands and macro commands follow.

GENERAL COMMANDS

open classname

Open the class called *classname* for further manipulation.

close [classname]

Close the open class called *classname.* It is an error if *classname* is not already opened. If no *classname* is given, then the currently open class is closed.

print

Print the currently open class.

insert [oid=oid_value] (value1 value2 ...)

Insert a new instance to the open class using *value1*, *value2*, etc., for its attribute values and *oid_value* for its OID. If *oid_value* is not "0", then this value will be used as the instance's object identifier. Otherwise, it is an error.

insert (value1 value2 ... **)**
As above, but the system generates a unique object identifier.

create classname **(** name1 = type1, name2 = type2, ... **)**
Create a class named *classname* with the attributes given in parentheses.

open (name1 = type1, name2 = type2,... **) as** classname
Open a class named *classname* for writing but do not record its existence in the system catalogs. (This is primarily to aid in bootstrapping.)

destroy classname
Destroy the class named *classname*.

define index index-name **on** class-name **using** amname
(opclass attr | function({attr}))
Create an index named *index_name* on the class named *classname* using the *amname* access method. The fields to index are called *name1*, *name2*, etc., and the operator collections to use are *collection_1*, *collection_2*, etc., respectively.

MACRO COMMANDS

define function macro_name **as** rettype function_name (args)
Define a function prototype for a function named *macro_name* which has its value of type *rettype* computed from the execution *function_name* with the arguments *args* declared in a C-like manner.

define macro macro_name **from file** filename
Define a macro named *macname* which has its value read from the file called *filename*.

EXAMPLE

The following set of commands will create the “pg_opclass” class containing the *int_ops* collection as object *421,* print out the class, and then close it.

```
create pg_opclass (opcname=char16)
open pg_opclass
insert oid=421 (int_ops)
print
close pg_opclass
```

SEE ALSO

initdb(unix), createdb(unix), createdb(commands), template(files).

NAME

$PGDATA/pg_hba — host-based access control

DESCRIPTION

POSTGRES95 provides host-based access control through the file **pg_hba**. This file is stored in $PGDATA and should have proper file permissions so that casual users cannot modify it. The file stores permissions information about which hosts are authorized to access databases.

The file consists of lines of three columns. The first column lists the name of the database. The second column lists the IP address of clients machines that are authorized to connect, and the third column provides a bit mask for the IP address. Lines which begin with '#' are considered comments.

The keyword "all" can be used to apply to all database names. Each column can contain strings of at most 80 characters, and each database may have up to 255 configuration lines that apply. (80 and 255 are constants that can be changed at POSTGRES95 compile time by altering the #defines MAX_TOKEN and MAX_LINES in src/backend/libpq/auth.c)

Below is the default pg_hba file that is installed.

```
#
# Example config file for Postgres95 host based access
#
# Lines starting with "all" apply to all databases.  Otherwise the first
# column has to match the name of the database being connected to.  Up to
# ten config lines can apply to each database.  Mask specifies bits that
# aren't counted. After those bits are taken out, the connection address
# must match the address in the middle column.
#
# <name>    <address>   <mask>
#
all         127.0.0.1   0.0.0.0
```

SEE ALSO

introduction(unix).

NAME

page structure — POSTGRES95 database file default page format

DESCRIPTION

This section provides an overview of the page format used by POSTGRES95 classes. User-defined access methods need not use this page format.

In the following explanation, a **byte** is assumed to contain 8 bits. In addition, the term **item** refers to data which is stored in POSTGRES95 classes.

Diagram 1 shows how pages in both normal POSTGRES95 classes and POSTGRES95 index classes (e.g., a B-tree index) are structured.

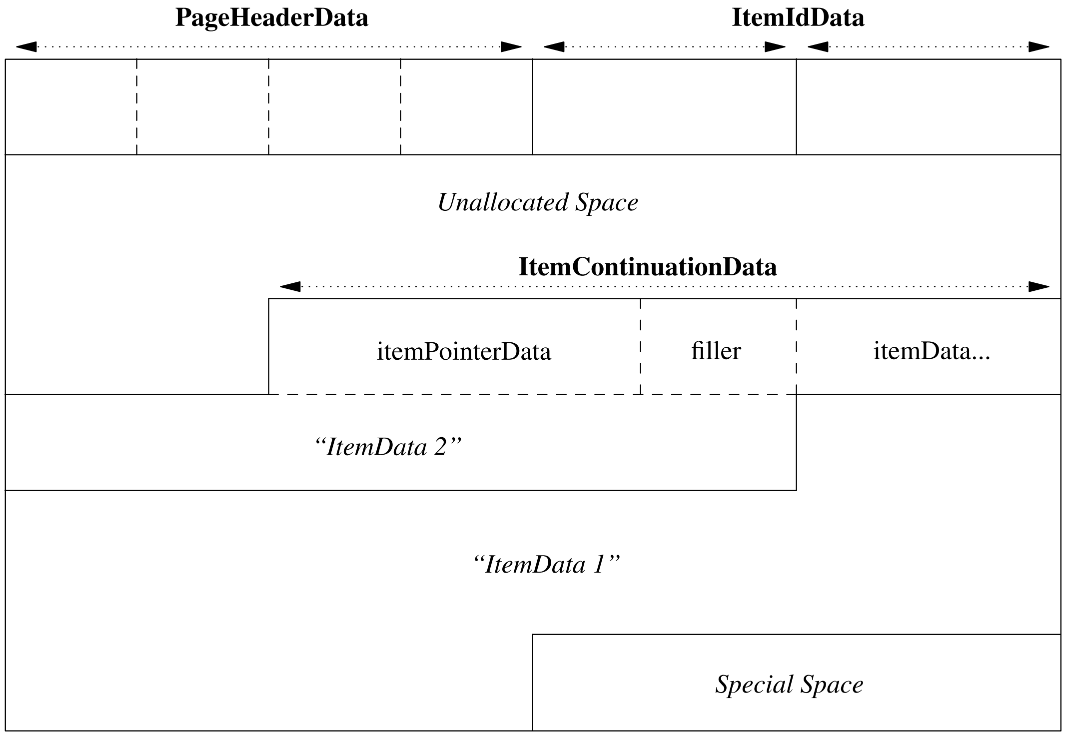

Diagram 1: Sample Page Layout

The first 8 bytes of each page consists of a page header (**PageHeaderData**). Within the header, the first three 2-byte integer fields, *lower*, *upper*, and *special*, represent byte offsets to the start of unallocated space, to the end of unallocated space, and to the start of "special space." Special space is a region at the end of the page which is allocated at page initialization time and which contains information specific to an access method. The last 2 bytes of the page header, *opaque*, encode the page size and information on the internal fragmentation of the page. Page size is stored in each page because frames in the buffer pool may be subdivided into equal sized pages on a frame by frame basis within a class. The internal fragmentation information is used to aid in determining when page reorganization should occur.

Following the page header are item identifiers (**ItemIdData**). New item identifiers are allocated from the first four bytes of unallocated space. Because an item identifier is never moved until it is freed, its index may be used to indicate the location of an item on a page. In fact, every pointer to an item (**ItemPointer**) created by POSTGRES95 consists of a frame number and an index of an item identifier. An item identifier contains a byte-offset to the start of an item, its length in bytes, and a set of attribute bits which affect its interpretation.

The items, themselves, are stored in space allocated backwards from the end of unallocated space. Usually, the items are not interpreted. However when the item is too long to be placed on a single page or when fragmentation of the item is desired, the item is divided and each piece is handled as distinct items in the following manner. The first through the next to last piece are placed in an item continuation structure **(**ItemContinuationData**).** This structure contains *itemPointerData* which points to the next piece and the piece itself. The last piece is handled normally.

FILES

.../data/...
: Location of shared (global) database files.

.../data/base/...
: Location of local database files.

BUGS

The page format may change in the future to provide more efficient access to large objects.

This section contains insufficient detail to be of any assistance in writing a new access method.

NAME

.../data/files/global1.bki — global database template
.../data/files/local1_XXX.bki — local database template
.../data/files/template1/* — default database template

DESCRIPTION

These files contain scripts which direct the construction of databases. Note that the "global1.bki" and "template1_local.bki" files are installed automatically when the POSTGRES95 super-user runs *initdb*. These files are copied from

```
.../src/backend/obj/{dbdb,local}.bki
```

The databases which are generated by the template scripts are normal databases. Consequently, you can use the terminal monitor or some other frontend on a template database to simplify the customization task. That is, there is no need to express everything about your desired initial database state using a BKI template script, because the database state can be tuned interactively.

The system catalogs consist of classes of two types: global and local. There is one copy of each global class that is shared among all databases at a site. Local classes, on the other hand, are not accessible except from their own database.

The file

```
.../data/files/global1.bki
```

specifies the process used in the creation of global (shared) classes by *createdb*. Similarly, the

```
.../files/local1_XXX.bki
```

files specify the process used in the creation of local (unshared) catalog classes for the "XXX" template database. "XXX" may be any string of 16 or fewer printable characters. If no template is specified in a *createdb* command, then the template in

```
.../files/local1_template1.bki
```

is used.

The .bki files are generated from C source code by an inscrutable set of AWK scripts.

BUGS

POSTGRES95 Version 1.01 does not permit users to have separate template databases.

SEE ALSO

bki(files), initdb(unix), createdb(unix).

REFERENCES

The following technical reports are referred to in this document. For information on ordering technical reports, see the installation notes that accompany the POSTGRES95 distribution.

[ONG90] Ong, L. and Goh, J., "A Unified Framework for Version Modeling Using Production Rules in a Database System," Electronics Research Laboratory, University of California, Berkeley, ERL Memo M90/33, April 1990.

[ROWE87] Rowe, L. and Stonebraker, M., "The POSTGRES Data Model," Proc. 1987 VLDB Conference, Brighton, England, Sept. 1987.

[SHAP86] Shapiro, L., "Join Processing in Database Systems with Large Main Memories," ACM-TODS, Sept. 1986.

[STON87] Stonebraker, M., "The POSTGRES Storage System," Proc. 1987 VLDB Conference, Brighton, England, Sept. 1987.

[STON90] Stonebraker, M. et. al., "On Rules, Procedures, Caching and Views in Database Systems," Proc. 1990 ACM-SIGMOD Conference on Management of Data, Atlantic City, N.J., June 1990.

[WONG76] Wong, E., "Decomposition: A Strategy for Query Processing," ACM-TODS, Sept. 1976.

REFERENCES(MANUAL)

REFERENCES(MANUAL)

Table of Contents

Section 1 — Introduction 1
Section 2 — UNIX Commands (UNIX) 2
General Information 2
Security 3
Cleardbdir 7
Createdb 8
Createuser 10
Destroydb 12
Destroyuser 14
Initdb 16
Ipcclean 17
monitor 18
pg_dump 21
postgres 22
postmaster 24
psql 27
Section 3 — What comes with POSTGRES95 (BUILT-INS) 30
Built-in and System Types 30
List of built-in types 30
Syntax of date and time types 31
Built-in operators and functions 31
Binary operators 32
Unary operators 37
Built-in aggregate functions 38
Section 4 — SQL Commands (COMMANDS) 39
General Information 39
Constants 39
Fields and Columns 40
Operators 41
Expressions 41
Commands 44
Abort 44
Alter Table 45
Begin 46
Close 47
Commit 48
Copy 49
Create Aggregate 52
Create Function 54
Create Index 60
Create Operator 63
Create Rule 66
Create 69
Create Type 71
Create Version 74
Create View 75
Create database 76
Delete 77
Drop database 78
Drop Table 79
Drop Aggregate 80
Drop Function 81
Drop Index 82
DROP Operator 83
Drop Rule 84
Drop Type 85
End 86
Fetch 87
Insert 88

Listen 90
Load 91
Notify 92
Purge 93
Drop View 94
Rename 95
Commands 96
Rollback 96
Select 97
Update 99
Vacuum 100
Section 5 — Libpq 101
Control and Initialization 101
Database Connection Functions 101
Query Execution Functions 102
Fast Path 104
Asynchronous Notification 105
Functions Associated with the COPY Command 105
LIBPQ Tracing Functions 106
User Authentication Functions 106
BUGS 107
Sample Programs 107
Section 6 — Large Objects 116
Historical Note 116
Inversion Large Objects 116
Large Object Interfaces 116
Creating a Large Object 116
Importing a Large Object 117
Exporting a Large Object 117
Opening an Existing Large Object 117
Writing Data to a Large Object 117
Seeking on a Large Object 117
Closing a Large Object Descriptor 117
Built in registered functions 117
Accessing Large Objects from LIBPQ 118
Sample Program 119
Section 7 — System Catalogs 124
Section 8 — Files 133
General Information 133
Backend Interface — BKI 134
Host based access control 136
Page Structure 137
Template 139
References 140